НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«На правах рукопису»
УДК 004.415.2

«До захисту допущено»
Науковий керівник кафедри
_____ І.А. Дичка
«___»_____2019 р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 121 Інженерія програмного забезпечення

на тему: «Метод створення документації для REST API на основі тестів»

Виконав:
студент ІІ курсу, групи КП-81мп
Сущик Андрій Миколайович _____

Керівник:
доцент кафедри ПЗКС, к.т.н., доцент,
Олещенко Любов Михайлівна _____

Нормоконтроль:
доцент кафедри ПЗКС, к.т.н., доцент,
Онай Микола Володимирович _____

Рецензент: доцент кафедри автоматики та управління
в технічних системах ФІОТ, к.т.н., доцент,
Полторак В.П. _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2019 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

Рівень вищої освіти – другий (магістерський) за освітньо-професійною програмою

Спеціальність (спеціалізація) – 121 «Інженерія програмного забезпечення» («Програмне забезпечення комп'ютерних та інформаційно-пошукових систем»)

ЗАТВЕРДЖУЮ

Науковий керівник кафедри

_____ І.А. Дичка

«___»_____ 2018 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Сущику Андрію Миколайовичу

1. Тема дисертації «Метод створення документації для REST API на основі тестів», науковий керівник дисертації доцент кафедри ПЗКС, к.т.н., доцент, Олещенко Любов Михайлівна затверджені наказом по університету від «13» листопада 2019 р. № 3895-С.

2. Термін подання студентом дисертації «17» грудня 2019 р.

3. Об'єкт дослідження: процес створення документації для REST API.

4. Предмет дослідження: методи створення документації для REST API.

5. Перелік завдань, які потрібно розробити:
   - проаналізувати вимоги до документації для REST API;
   - проаналізувати методи створення документації для REST API;
   - розробити інтерфейс та архітектуру методу;
   - розробити реалізацію методу у вигляді бібліотеки;
   - обґрунтувати вибір технологій для реалізації;
   - порівняти ефективність запропоного методу з іншими методами.

6. Орієнтовний перелік публікацій:
   - XII науково-практична конференція магістрантів та аспірантів "ПМК-2019"

7. Дата видачі завдання «15» грудня 2018 р

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Онай М.В., к.т.н., доцент		

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1.	Ґрунтовне ознайомлення з предметною галуззю	17.02.2019	
2.	Визначення структури магістерської дисертації; вивчення літератури, пошук додаткової літератури, патентний пошук	04.03.2019	
3.	Робота над першим розділом магістерської дисертації; проведення наукового дослідження	16.04.2019	
4.	Проведення наукового дослідження; робота над другим розділом магістерської дисертації; розроблення програмного забезпечення	14.05.2019	
5.	Проведення наукового дослідження; робота над статтею за результатами наукового дослідження	17.09.2019	
6.	Проведення наукового дослідження; робота над третім розділом магістерської дисертації; підготовка матеріалів доповіді на конференції ПМК-2019.	02.10.2019	
7.	Завершення роботи над основною частиною магістерської дисертації; підготовка ілюстративного матеріалу;	28.10.2019	
8.	Оформлення текстової і графічної частини магістерської дисертації	10.11.2019	

Студент А.М. Сущик

Науковий керівник дисертації Л.М. Олещенко

РЕФЕРАТ

Актуальність теми. Під час розроблення та тестування програмного забезпечення більшість часу витрачається на аналіз коду, доменної логіки, написання тестів та документації. Тести та документація є важливими артефактами, і якщо вони якісні, то суттєво зменшують часові витрати розробників на аналіз доменної логіки та аналіз програмного коду.

При розробленні клієнт-серверного програмного забезпечення важливою є якість документації до API, що надають доступ до даних, адже в деяких випадках вона є механізмом, за допомогою якого організовується взаємодія між командами клієнтської та серверної частини програмного забезпечення.

Створення та підтримка якісної документації за допомогою наявних методів потребує багато часу, тому сьогодні проблема оптимізації та пошуку нових методів для створення документації до API є дуже актуальною.

Об'єктом дослідження є процес створення документації для REST API.

Предметом дослідження є методи створення документації для REST API.

Мета дослідження: оптимізувати та автоматизувати процес створення документації для REST API.

Наукова новизна полягає в тому, що вперше запропоновано метод створення документації для REST API на основі тестів з використанням стандарту OpenAPI.

Практична цінність результатів, що були отримані в роботі, полягає в тому, що запропонований метод створення документації для REST API дозволяє витрачати на створення документаціменше часу, надає для користувачів інтерактивний графічний інтерфейс та суттєво зменшує ймовірність помилок в документації, які спричинені людським фактором.

Апробація роботи. Основні положення і результати роботи були представлені та обговорювались на XII науковій конференції магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2019 (м. Київ).

Структура та обсяг роботи. Магістерська дисертація складається з вступу, п'ятьох розділів, висновків та додатків.

У вступі наведений сучасний стан проблеми та обґрунтована її актуальність.

У першому розділі проаналізовані стандарти створення документації для програмного забезпечення та обґрунтована актуальність проблеми для стандарту REST API, сформовані вимоги до документації, проаналізовані наявні методи створення документації, виявлено їх основні переваги та недоліки та відповідність сформованим вимогам.

У другому розділі розглянута наявна реалізація методу створення документації на основі тестів, виявлено її основні недоліки та запропоновано новий метод.

У третьому розділі наведені інтерфейс та архітектура запропонованого методу, обґрунтовано вибір технологій для програмної розробки.

У четвертому розділі розглянуті особливості тестування методу та проведене порівняння його ефективності в порівнянні з іншими методами та наявною реалізацією.

У п'ятому розділі наведений спосіб комерціалізації методу. В створеній бізнес-моделі розглянуто проблему предметної області, сформовані зацікавлені сторони, комерційне рішення та його основні характеристики, конкурентні переваги, клієнти та сегменти ринку, унікальна ціннісна пропозиція, здійснено аналіз доходів та витрат.

У висновках проаналізовано отримані результати роботи.

У додатках наведено копія презентації та лістинги основних класів.

Робота виконана на 82 аркушах, містить 2 додатки та посилання на список використаних літературних джерел з 40 найменувань. У роботі наведено 18 рисунків та 7 таблиць.

Ключові слова: документація, REST API, тести.

ABSTRACT

Topicality. Most of the time spent developing and testing software is spent on code analysis, domain logic, test writing, and documentation. Tests and documentation are important artifacts, and if they are of high quality, then significantly reduce developers' time spent on domain logic analysis and code analysis.

When developing client-server software, the quality of documentation to the APIs that provides access to the data is important, because in some cases it is the mechanism by which the client-server and server-side software commands are organized.

Creating and maintaining quality documentation using existing methods takes a long time, so today the problem of optimizing and finding new methods to create API documentation is very important.

The object of research is the process of creating documentation for the REST API.

The subject of the study are methods for creating documentation for the REST API.

The aim of the study is to optimize and automate the documentation creation process for the REST API.

The scientific novelty. For the first time a method for creating documentation for REST APIs based on tests using the OpenAPI standard is proposed.

The practical value of the results obtained in the work is that the proposed method for creating documentation for the REST API allows you to spend on document creation in less time, provides users with an interactive graphical interface, and significantly reduces the likelihood of human-caused documentation errors.

Test work. The main provisions and results of the work were presented and discussed at the XII Scientific Conference of Undergraduate and Graduate Students in Applied Mathematics and Computing PMK-2019 (Kyiv).

Structure and scope of work. The master's dissertation consists of an introduction, five sections, conclusions and appendices.

The introduction provides a general description of the work, an assessment of the current state of the problem is made, the relevance of the research direction is substantiated, information about testing the results is given.

The first section analyzes the documentation standards for the software and substantiates the relevance of the problem for the REST API standard, the requirements for the documentation are formed, the available methods of documentation creation are analyzed, their main advantages and disadvantages and compliance with the formed requirements are revealed.

The second section discusses the existing implementation of the test-based documentation creation method, identifies its major drawbacks, and proposes a new method.

The third section presents the interface and architecture of the proposed method, justifies the choice of technologies for software development.

The fourth section discusses the peculiarities of testing the method and comparing its effectiveness with other methods and the existing implementation.

The fifth section describes how to commercialize the method. In the created business model the problem of the domain is considered, the formed stakeholders, the commercial decision and its main characteristics, competitive advantages, clients and market segments, unique value proposition, the analysis of income and expenses.

The results of the work are analyzed in the conclusions.

Attachments provide a copy of the presentation and class listings.

The work is made on 82 sheets, contains 2 appendices and links to the list of used literary sources of 40 titles. The paper presents 18 figures and 7 tables.

Keywords: documentation, REST-API, tests.

РЕФЕРАТ

Актуальность темы. При разработке и тестированию программного обеспечения большую часть времени тратится на анализ кода, доменной логики, написание тестов и документации. Тесты и документация являются важными артефактами, и если они качественные, то существенно уменьшают временные затраты разработчиков на анализ доменной логики и анализ программного кода.

При разработке клиент-серверного программного обеспечения важно качество документации к API, предоставляющих доступ к данным, ведь в некоторых случаях она является механизмом, с помощью которого организуется взаимодействие между командами клиентской и серверной части программного обеспечения.

Создание и поддержка качественной документации с помощью имеющихся методов требует много времени, поэтому сегодня проблема оптимизации и поиска новых методов для создания документации к API является очень актуальной.

Объектом исследования является процесс создания документации для REST API.

Предметом исследования являются методы создания документации для REST API.

Цель исследования: оптимизировать и автоматизировать процесс создания документации для REST API.

Научная новизна заключается в том, что впервые предложен метод создания документации для REST API на основе тестов с использованием стандарта OpenAPI.

Практическая ценность результатов, полученных в работе, заключается в том, что предложенный метод создания документации для REST API позволяет тратить на создание документациименше времени, предоставляет для пользователей интерактивный графический интерфейс и

существенно уменьшает вероятность ошибок в документации, вызванных человеческим фактором.

Апробация работы. Основные положения и результаты работы были представлены и обсуждались на XII научной конференции магистрантов и аспирантов «Прикладная математика и компьютинг» ПМК-2019 (г.. Киев).

Структура и объем работы. Магистерская диссертация состоит из введения, пяти глав, заключения и приложений.

Во введении приведен современное состояние проблемы и обоснована ее актуальность.

В первой главе проанализированы стандарты создания документации для программного обеспечения и обоснована актуальность проблемы для стандарта REST API, сформированы требования к документации, проанализированы существующие методы создания документации, выявлены их основные преимущества и недостатки и соответствие сложившимся требованиям.

Во втором разделе рассмотрена существующая реализация метода создания документации на основе тестов, выявлено ее основные недостатки и предложен новый метод.

В третьем разделе приведены интерфейс и архитектура предложенного метода, обоснован выбор технологий для программной разработки.

В четвертом разделе рассмотрены особенности тестирования метода и проведено сравнение его эффективности по сравнению с другими методами и имеющейся реализацией.

В пятом разделе приведен способ коммерциализации метода. В созданной бизнес-модели рассмотрена проблема предметной области, сформированы заинтересованные стороны, коммерческое решение и его основные характеристики, конкурентные преимущества, клиенты и сегменты рынка, уникальная ценностная предложение, осуществлен анализ доходов и расходов.

В выводах проанализированы полученные результаты работы.

В приложениях приведены копия презентации и листинги основных классов.

Работа выполнена на 82 листах, содержит 2 приложения и ссылки на список использованных литературных источников из 40 наименований. В работе приведены 18 рисунков и 7 таблиц.

Ключевые слова: документация, REST API, тесты.

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

JSON – JavaScript Object Notation, текстовий формат обміну даними між комп'ютерами.

XML – Extensible Markup Language, стандарт побудови мов розмітки ієрархічно структурованих даних для обміну між різними застосунками, зокрема, через мережу Інтернет.

Кешування – швидкісна пам'ять або частина оперативної пам'яті, де зберігаються копії часто використовуваних даних. Забезпечує швидкий доступ до них. Кеш-пам'ять зберігає вміст і адреси даних, до яких часто звертається процесор.

HTTP – Hyper Text Transfer Protocol, протокол передачі даних, що використовується в комп'ютерних мережах.

FTP – File Transfer Protocol, протокол передачі файлів по мережі.

SMTP – Simple Mail Transfer Protocol, комунікаційний протокол для пересилання електронної пошти.

URI – Uniform Resource Identifier, компактний рядок літер, який однозначно ідентифікує окремий абстрактний чи фізичний ресурс.

Юзабіліті – поняття в мікроергономіці, що визначає загальну ступінь зручності предмета при використанні.

YAML – призначений для читання людиною формат серіалізації даних, концептуально близький до мов розмітки, але орієнтований на зручність введення-виведення типових структур даних багатьох мов програмування.

Markdown-розмітка – полегшена мова розмітки даних, яку створено з ухилом на зручність у публікації з подальшим перетворенням її на HTML.

Роутинг – процес, який відповідає за визначення обробки для конкретної заданої сторінки.

Система контролю версій – програмний інструмент для керування версіями одиниці інформації: вихідного коду програми, скрипту, веб-сторінки, веб-сайту, 3D-моделі, текстового документу тощо.

Неперервна інтеграція – практика розроблення програмного забезпечення, яка полягає у виконанні частих автоматизованих складань проекту для якнайшвидшого виявлення та вирішення інтеграційних проблем.

Шаблонізатор – програмне забезпечення, призначене для поєднання шаблонів із моделлю даних для створення результативних документів.

Верифікація – процедура документального підтвердження особистих даних користувача.

Epub – відкритий стандарт формату електронних книг.

СУБД – система управління базами даних, набір взаємопов'язаних даних і програм для доступу до цих даних.

MVC – Model-view-controller, архітектурний шаблон, який використовується під час проектування та розроблення програмного забезпечення.

Фреймворк – інфраструктура програмних рішень, що полегшує розроблення складних систем.

ORM – Object-relational mapping, це технологія програмування, яка зв'язує бази даних з концепціями об'єктно-орієнтованих мов програмування, створюючи «віртуальну об'єктну базу даних».

DDD – Domain-driven design, підхід до моделювання складного об'єктно-орієнтованого програмного забезпечення.

MIME-тип – код, який визначає формат файлу або тип контенту, що передається мережею Інтернет.

URL – Uniform Resource Locator, стандартизована адреса певного ресурсу (такого як документ, чи зображення) в інтернеті (чи деінде).

Web Hook – метод збільшення або розширення функціональності веб-сторінки або веб-застосунку за допомогою користувацьких зворотних викликів.

ВСТУП

Програмісти витрачають відносно невелику кількість часу на написання або зміну існуючого коду. Більшість часу відводиться для аналізу коду і доменної логіки, написання тестів та документації. Тести та документація є важливими артефактами, і якщо вони якісні, то суттєво зменшують часові затрати розробників на аналіз доменної логіки та аналіз програмного коду.

Зокрема, при розробленні клієнт-серверного програмного забезпечення важливою є якість документації до API, що надають доступ до даних. Часто клієнтська та серверна частини програмного забезпечення розробляються різними розробниками окремо на великій відстані та в різних часових поясах. У таких випадках документація є механізмом, за допомогою якого організовується взаємодія між командами клієнтської та серверної частини програмного забезпечення, тому її якість є критично важливою.

Створення та підтримка якісної документації за допомогою наявних методів потребує багато часу, тому сьогодні проблема оптимізації та пошуку нових методів для створення документації до API є дуже актуальною.

1. АНАЛІЗ ПРОЦЕСУ СТВОРЕННЯ ДОКУМЕНТАЦІЇ ДЛЯ МЕРЕЖЕВИХ ПРОГРАМНИХ ПРИКЛАДНИХ ІНТЕРФЕЙСІВ

1.1. Порівняння процесу створення документації в залежності від стандарту

На даний момент для створення програмних прикладних інтерфейсів найчастіше використовуються стандарти: GraphQL, JSON RPC, SOAP і REST API.

1.1.1. GraphQL

GraphQL – це стандарт декларування структури і способів отримання даних або синтаксис, який описує, як можна зчитувати дані з серверу [1]. GraphQL має три основні характеристики:

- дозволяє клієнту точно вказати, які дані йому потрібні;
- полегшує агрегацію даних з декількох джерел;
- використовує систему типів для опису даних.

При такому підході, крім гнучкості, зменшується кількість запитів та обсяг даних на транспортному рівні. GraphQL API базується на трьох основних будівельних блоках: схемі (schema), запитах (queries) і розпізнавачах (resolvers). У GraphQL передбачені такі види операцій: запит (зчитування даних), мутація (запис даних) або підписка (безперервне зчитування даних) [2]. Будь-яка з таких операцій – просто рядок, який необхідно зібрати відповідно до специфікації мови запитів GraphQL. Як тільки така операція прийде на сервер з клієнтської програми, її можна буде інтерпретувати за допомогою усієї схеми GraphQL і віддати дані, яких потребує клієнтський застосунок. GraphQL може працювати з будь-яким високорівневим мережевим протоколом (найчастіше використовується HTTP) та з будь-яким форматом даних (зазвичай використовується JSON).

Перевагою використання GraphQL є декларативність. Також, до переваг відносять сильну та чітку типізацію, відсутність проблем з

версіонуванням. Ще один важливий аспект GraphQL – його ієрархічний характер. GraphQL побудований на взаємозв'язку між об'єктами, що спрощує формування запитів, де службі RESTful може знадобитися багаторазова система запитів «request/response» або складна операція об'єднання в SQL [3]. Головним недоліком прийнято вважати складність реалізації на серверній частині. Зазвичай, саме з цієї причини GraphQL використовують як додатковий шар між клієнтом і веб-сервісами. При цьому веб-сервіси не використовують GraphQL, а надають доступ до даних за допомогою REST API.

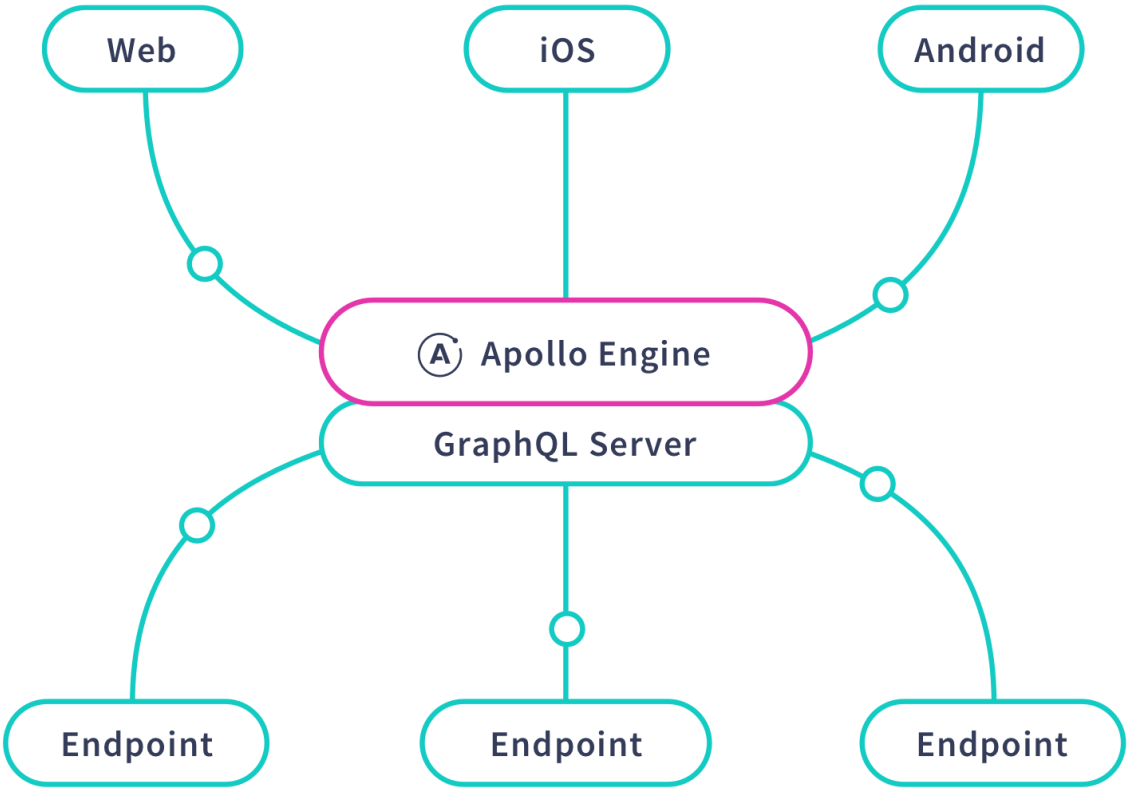


Рис. 1.1. Типове використання GraphQL в мікросервісній архітектурі

Завдання автоматизації та оптимізації процесу створення документації є неактуальним, адже вищезгадана GraphQL-схема і є документацією для користувачів мережевих програмних прикладних інтерфейсів. Більше того, вже створено GraphiQL – кросплатформений інструмент для розробки застосунків з використанням GraphQL [4], графічний інтерфейс якого зображений на рис. 1.2.

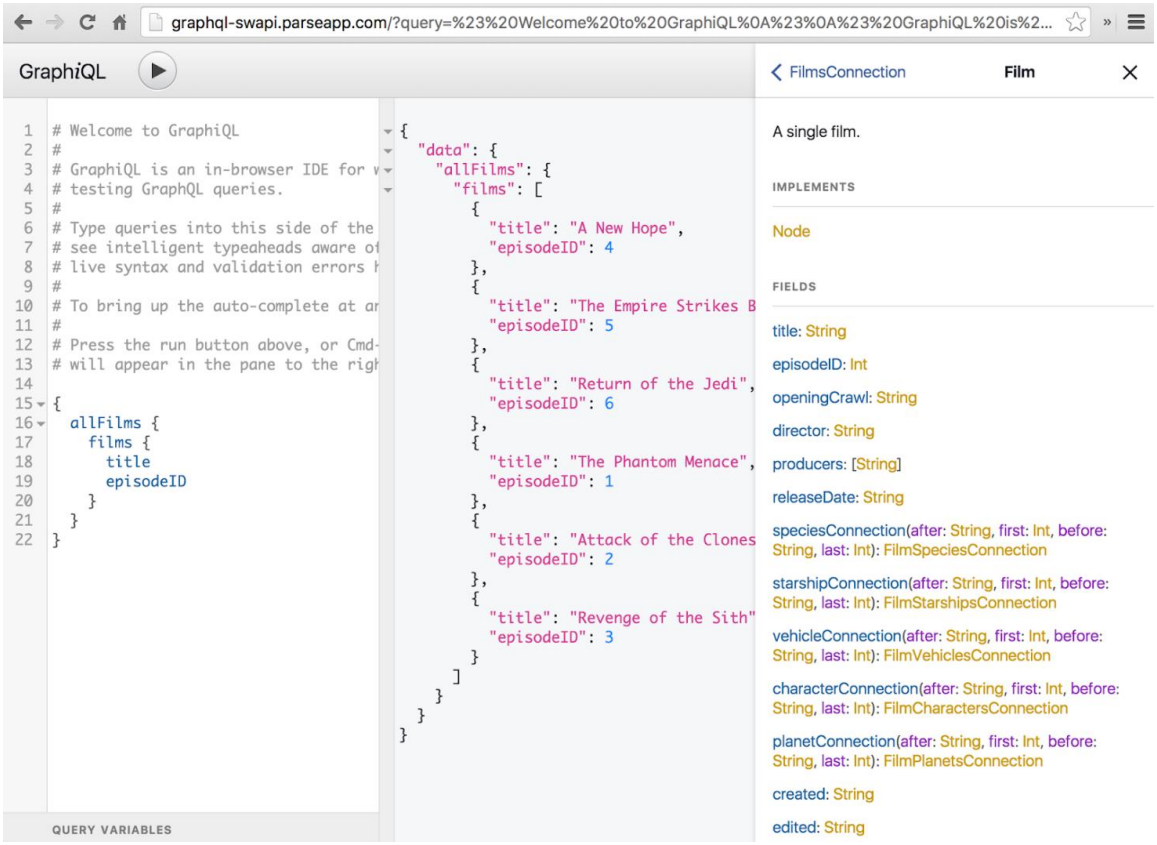


Рис. 1.2. Графічний інтерфейс інструменту GraphiQL

За допомогою цього інструменту програміст має можливість переглянути документацію в читабельному вигляді, яка буде автоматично згенеровано на основі схеми. Крім того, переглядаючи документацію, можна відправляти запити або мутації та відразу переглядати результат в зручній формі.

1.1.2. JSON-RPC

JSON-RPC – це протокол віддаленого виклику процедур, який використовує JSON в якості формату даних. Даний протокол багато в чому схожий на XML-RPC, його специфікація визначає кілька типів даних і правила їх обробки [5]. Він розроблений, як простий, гнучкий та зрозумілий для всіх стандарт. JSON-RPC базується на відсиланні запитів до сервера, який реалізує віддалений протокол.

Всі дані, що передаються – запити, серіалізовані в JSON. Запит – виклик певного методу, наданого віддаленою системою. Він повинен містити три обов'язкових компоненти:

- «method» – рядок з назвою методу;
- «params» – дані, які передаються методу як параметри;
- «id» – значення будь-якого типу, яке використовується для встановлення відповідності між запитом і відповіддю.

Сервер повинен повернути відповідь на кожен отриманий запит. Відповідь повинна містити такі властивості:

- «result» – дані, які повертає метод. Якщо сталася помилка під час виконання методу, ця властивіст повинна бути встановлена в null;
- «error» – код помилки у випадку помилки під час виконання методу, інакше null;
- «id» – те ж саме значення, що і в запиті, до якого відноситься дана відповідь.

Д ля ситуацій, коли відповідь не потрібна, були введені нотифікації. Нотифікації відрізняються від запиту відсутністю «id».

Як основну перевагу JSON-RPC можна відзначити його простоту та інтуїтивність. Часто при розробленні API програмісти, які нічого не знають про стандарти, самі проектують інтерфейси зі схожою структурою запитів та відповідей. JSON-RPC добре підходить для веб-сервісів з невеликим обсягом функціональності та типів даних. Проте, нестача механізмів кешування та версіонування, відсутність чіткої специфікації роблять даний стандарт непридатним для об’ємних веб-сервісів.

Стандарт JSON-RPC є дуже простим, тому простою є і задача генерування документації для API, які використовують цей стандарт. Зокрема, все, що повинна містити документація – це список методів, параметрів, відповідей та кодів помилок. З цієї задачею добре справляються реалізації цього стандарту для різних платформ. Зокрема, це – JSON-RPC.NET для платформи .NET, go/net/rpc для GoLang, php-json-rpc для PHP [6].

1.1.3. SOAP

SOAP – протокол обміну структурованими повідомленнями в розподілених обчислювальних системах [7]. Для протоколу SOAP не існує різниці між викликом процедури і відповіддю на виклик, він просто визначає формат послання (message) у вигляді XML-документу. Послання може містити виклик процедури, відповідь на нього, запит на виконання якихось інших дій. Специфікація SOAP не використовує аналіз вмісту послання, вона задає тільки його стандарт для його оформлення. SOAP заснований на мові XML і розширює один з протоколів прикладного рівня – HTTP, FTP, SMTP і т.д. Як правило, найчастіше використовується HTTP. SOAP-повідомлення є XML-документом, який складається з трьох основних елементів: конверт (SOAP Envelope), заголовок (SOAP Header) і тіло (SOAP Body).

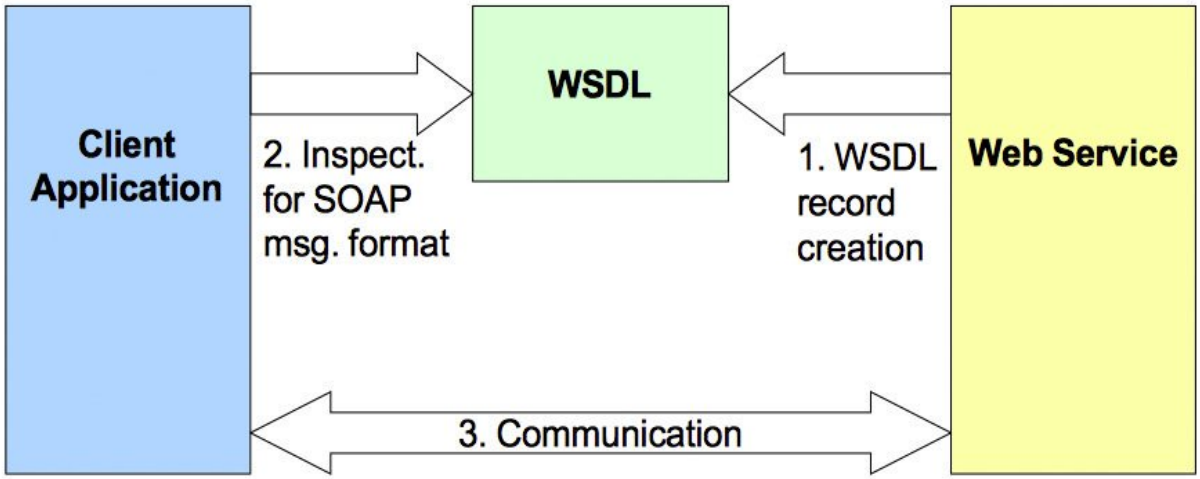


Рис. 1.3. Обмін повідомленнями між клієнтом і сервером з використанням SOAP

До переваг використання стандарту SOAP можна віднести його незалежність від транспортного протоколу та дуже чітку специфікацію. Водночас, недоліками є його складність та негнучкість, а також обмеження форматом XML, який є доволі об’ємним з точки зору транспортування.

Для створення документація для SOAP-API використовують WSDL-мову визначення інтерфейсу сервісу, яка заснована на форматі XML, що

описує функціональність сервісу і спосіб доступу до нього [8]. Цю мову використовують як для створення документації, так і для генерації коду, тому задача створення документації для SOAP-API є неактуальною.

1.1.4. REST API

REST – підхід до архітектури мережевих протоколів, які забезпечують доступ до інформаційних ресурсів [9]. Був описаний і популяризований Роєм Філдінгом, одним із творців протоколу HTTP. Філдінг розробив REST паралельно з HTTP 1.1 базуючись на попередній версії 1.0 [10].

REST, як і кожен архітектурний стил, має декілька архітектурних обмежень. Одне з обмежень – це клієнт-серверна архітектура. Архітектура такого типу вимагає розділення відповідальності між компонентами, які займаються зберіганням та оновленням даних (сервером), і тими компонентами, які займаються відображенням даних на інтерфейсі користувача та реагування на дії з цим інтерфейсом (клієнтом) [11]. Таке розділення дозволяє компонентам працювати незалежно. Наступним обмеженням є те, що взаємодії між сервером та клієнтом не мають стану, тобто кожен запит містить всю необхідну інформацію для його обробки, і не покладається на те, що сервер знає щось з попереднього запиту. Додатковим обмеженням стилю REST є те, що системи, написані в цьому стилі, повинні підтримувати кешування, тобто дані, які передаються сервером, повинні містити інформацію про те, чи можна їх кешувати, і якщо можна, то як довго. Це дозволяє збільшувати продуктивність, уникаючи зайвих запитів, але також зменшує надійність системи через те, що дані в кеші можуть бути застарілими.

На практиці, REST API – це набір URI, HTTP викликів до цих URI та велика кількість представлень ресурсів у форматі JSON або XML, багато з яких будуть містити перехресні посилання. За основу адресації береться покриття ресурсів унікальними ідентифікаторами. Обмеження одноманітності інтерфейсу частково реалізовано за допомогою комбінацій

URI і HTTP дієслів і їх використанням відповідно до стандартів і конвенцій. Ресурси повинні бути іменниками, а дія над ресурсом – це дієслово. URI завжди повинен посилатися на ресурс, а не на дію.

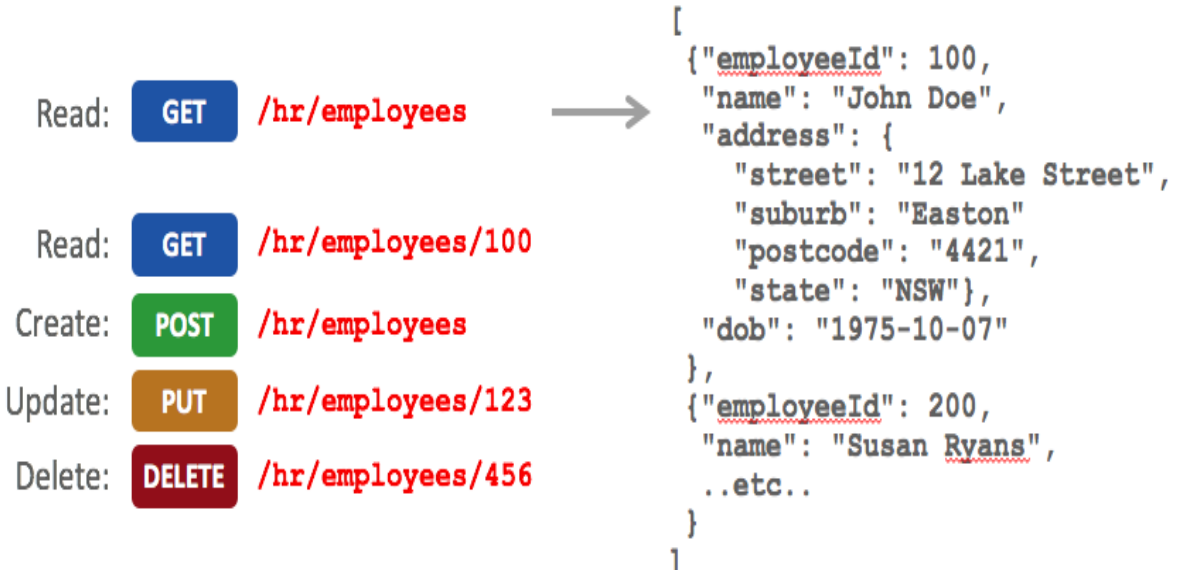


Рис. 1.4. Типовий ресурс та набір URI для доступу до нього

Кожен ресурс сервісу повинен мати хоча б один URI, який його ідентифікує. URI повинні мати просту ієрархічну структуру, щоб полегшити розуміння API, і як наслідок, його юзабіліті.

Документація до REST API повинна містити набір всіх ресурсів, їх ідентифікаторів та представлень, а також набір URI та HTTP-дієслів для доступу до ресурсів, інформацію про авторизацію, можливі коди помилок та відповідей. Сама специфікація REST API не передбачає ніяких автоматичних механізмів для створення документації, як це відбувається при використанні GraphQL за допомогою схеми, або в SOAP за допомогою WSDL. Водночас документація до REST API є важливим артефактом для клієнтської частини, а процес її створення та підтримки можна оптимізувати, детально проаналізувавши вимоги та методи.

1.2. Аналіз вимог до документації для REST API

Найголовніша вимога до будь-якої технічної документації – її актуальність. У випадку коли для розроблення програмного забезпечення

використовується каскадна модель цю вимогу дуже легко задовольнити. Але на сьогоднішній день (особливо в сфері бізнесу) мало хто використовує цю модель, натомість надають перевагу гнучким моделям, в яких доволі динамічно змінюються вимоги до програмного забезпечення. В такому випадку документацію потрібно оновлювати так само динамічно, як і вносити зміни в програмне забезпечення. Особливо гостра ця проблема для документування API. В переважній більшості випадків над клієнтською та серверною частиною працюють окремі команди – інколи вони знаходяться територіально далеко один від одного та не мають можливості взаємодіяти між собою. Саме тому документація є механізмом, за допомогою якого організовується взаємодія між цими командами, і її актуальність та якість є важливим фактором, який впливає на швидкість розроблення та якість кінцевого продукту.

Створенням та підтримкою документації для API може займатися окремий співробітник. Він буде поширювати цю документацію у вигляді файлів або публікувати на сайті. Такий підхід має ряд істотних недоліків з точки зору людського фактору та ресурсозатратності. Людина може щось забути, написати неправильно і т.д. Як тільки код змінився – документація застаріла. З неактуальною і помилковою документацією API-інтерфейсів розробники стикаються дуже часто, це забирає додатковий час, що в кінцевому підсумку позначається на вартості продуктів. Отже, під актуальністю мається на увазі усунення «людського фактору». Документація повинна в автоматичному або напівавтоматичному режимі оновлюватись разом зі змінами в роботі API, при цьому можливість забути внести зміни в документацію повинна бути зведена до мінімуму.

Наступна вимога до документації – її інтерактивність. Це означає, що ми повинні надавати певний користувацький інтерфейс, який не тільки в читабельному вигляді описує API, а й дозволяє виконувати запити до серверної частини. Це важливо, тому що азвичай користувачі API, які є розробниками клієнтської частини, перед тим, як писати код, все одно

роблять запити до серверної частини для того, щоб або перевірити працездатність API, або подивитись як поводить себе API в ситуаціях, які не описані в документації. За допомогою графічного інтерфейсу ми позбавляємо розробників необхідності користуватись для таких цілей стороннім програмним забезпеченням на кшталт Postman. Також, якщо у процесі розроблення програмного забезпечення беруть участь мануальні тестувальники, то вони зможуть окремо тестувати серверну частину та їм простіше буде «знайти крайнього» (клієнтську або серверну частину) у випадку, коли робота програми не відповідає поставленим вимогам, адже тестувальники за допомогою інтерактивної документації зможуть перевірити коректність роботи серверної частини автономно. Ще одна перевага інтерактивності – це те, що вони фактично нівелюють ситуації, коли в документації не до кінця описані можливі відповіді або коди помилок, адже в такому випадку без особливих зусиль користувачі API зможуть самі перевірити такі ситуації.

У випадку REST API для того, щоб забезпечити інтерактивність документації, немає потреби самому розробляти графічний інтерфейс, адже для візуалізації REST API вже існує багато готових бібліотек для різних платформ. Принцип їх роботи однаковий незалежно від мови програмування чи платформи, яка використовується. Ці бібліотеки зчитують файл, структура та формат якого повинні відповідати певній специфікації. Найчастіше для цього використовуються такі специфікації як RAML, APIBluePrint та OpenAPI, більш відома як Swagger [12]. Саме OpenAPI є найбільш поширеною та володіє найбільшою кількістю функціональності. За розробкою цієї специфікації наглядає Open API Initiative, проект Linux Foundation з 2010 року і активно вдосконалює її по сьогоднішній день [13]. Як формат для файлу може використовуватись XML, JSON або YAML [14]. Натомість API BluePrint обмежена файлом з markdown-розміткою, і не забезпечує інтерактивність [15]. Специфікація RAML немає підтримки серед такої кількості платформ та мов

програмування та менше знайома розробникам [16]. Саме тому, найкращий спосіб забезпечити інтерактивність – використати стандарт OpenAPI для опису роботи веб-сервісу та бібліотеки для візуалізації та інтерактивності в залежності від технологій, які використовуються для розроблення веб-сервісу.

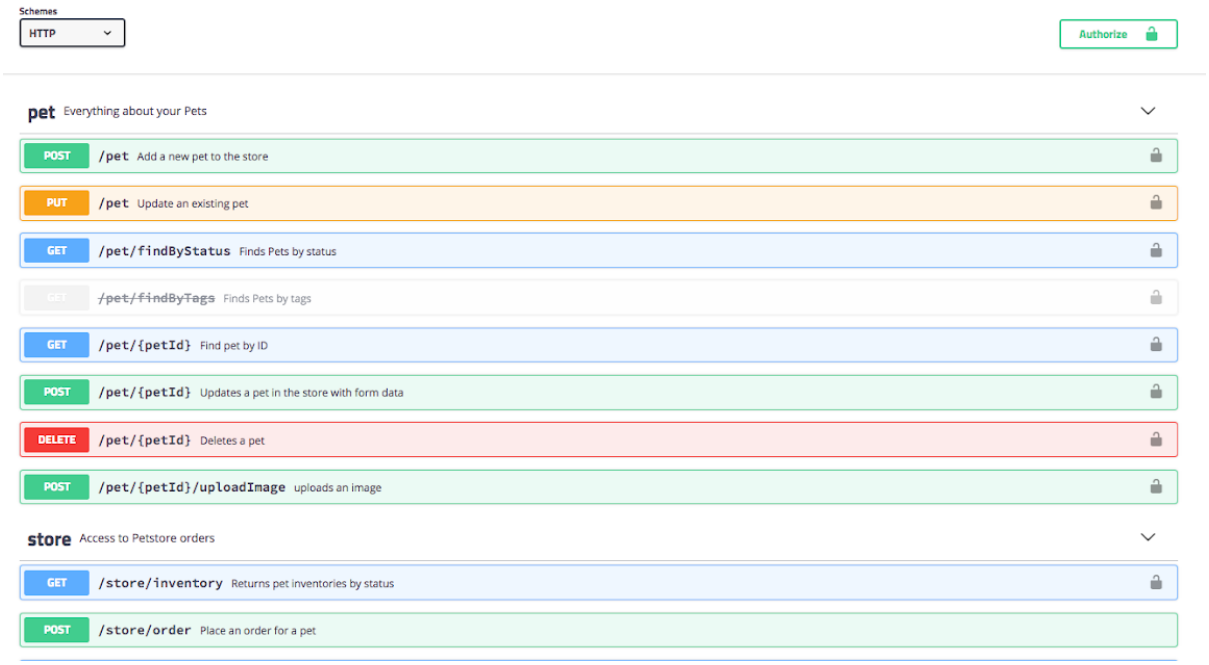


Рис. 1.5. Інтерактивна документація для REST API, створена з використанням специфікації OpenAPI

Також важлива вимога до документації – мінімальні часові затрати на її створення та підтримку. Ця вимога не потребує особливої аргументації, більше того, у випадку документації для API часто саме на цей критерій і звертають найбільше уваги при виборі методу для створення документації. Адже є випадки, коли її якість грає формальну роль – наприклад, коли над клієнтською і серверною частиною працює одна і та ж команда, або команди, які мають можливість швидкої комунікації між собою. В такому разі ціль документації – отримати суху інформацію – назви полів моделей, параметрів, методів тощо.

1.3. Аналіз методів створення документації для REST API

Для створення та підтримки документації для REST API найчастіше використовують такі методи:

- створення документації за допомогою інструментів для конкретної мови програмування/фреймворку;
- написання документації вручну;
- створення документації за допомогою сторонніх утиліт;
- створення документації на основі тестів.

1.3.1. Створення документації за допомогою інструментів для конкретної мови програмування/фреймворку

Принцип дії цих інструментів в переважній більшості однаковий. Зазвичай, вони встановлюються як бібліотека для розробницького оточення, а генерація документації відбувається шляхом запуску бінарного файлу або консольної команди. Консольна команда аналізує роутинг, анотації або файли в залежності від мови програмування та фреймворку, які використовуються у проекті. Її результатом є Swagger-файл або файли в форматі HTML та Javascript. Дані інструменти можуть надавати також додаткові функціональні можливості – зокрема, генерація колекції для Postman, генерація клієнтського коду тощо.

Можливість мінімізації людського фактору залежить від принципу роботи конкретного інструменту. Деякі з них зчитують анотації, які написані розробниками спеціально для генерації документації і не впливають безпосередньо на роботу програми. Зокрема, так працює бібліотека NelmioApiDocBundle для Symfony [17]. Цей принцип не дає можливості усунути людський фактор та засмічує код зайвими коментарями. Натомість, інші працюють з роутингом та анотаціями до класів, які впливають на те, як виконується код. Щоправда, при такому підході не завжди є можливість ручного редагування кінцевої документації – наприклад, додавання коментарів для моделей та запитів.

Більшість таких інструментів не генерують інтерактивну документацію. Як було сказано вище, зазвичай, результатом їх роботи є файли, які можуть інтерпретувати і відобразити в людино-читабельному вигляді браузер. Це призводить до того, що користувачі та тестувальники встановлюють та використовують сторонні програми для перевірки роботи API.

Даний спосіб потребує найменше часових затрат серед усіх вище перелічених запитів за умови наявності готових інструментів для технологій, які використовуються для написання серверної частини, адже достатньо лише встановити бібліотеку та налаштувати автоматичне оновлення документації після внесення змін в роботу програмної частини за допомогою системи контролю версій та системи неперервної інтеграції.

1.3.2 Написання документації вручну

Даний метод передбачає створення та підтримки документації вручну за допомогою markdown-розмітки або використання .docx-формату. Цей метод не потребує вивчення нових бібліотек чи утиліт, не вимагає знання стандартів та налаштування процесу неперервної інтеграції. Він доволі часто використовується для невеликих проектів.

Якість та відповідність документації програмного забезпечення повністю залежить від людини або групи людей, яка займається розробкою та підтримкою документації до API. Це найголовніший недолік даного методу, адже через людський фактор є велика можливість того, що документація не відповідає дійсності.

Метод написання документації вручну не забезпечує інтерактивний графічний інтерфейс. Натомість, є можливість обрати для документації зручний формат та структуру, і таким чином, покращити її якість та читабельність.

Також даний метод потребує багато часових затрат за умови, що не використовується шаблонізатор. Даний недолік сприяє тому, що цей метод обирають доволі рідко і надають перевагу іншим.

Також до недоліків даного методу можна віднести складність побудови процесу внесення змін до документації. Якщо для цього використовується система контролю версій, то існує велика ймовірність конфліктів при паралельному внесенні змін. При використанні звичайних текстових редакторів ця проблема нікуди не зникає, більше того, з'являється проблема її поширення між членами команди. Для використання хмарних технологій на кшталт Google Docs, зазвичай, потрібна реєстрація та налаштування доступу до необхідних документів, що потребує додаткового часу, наприклад, при вступі нових людей в команду.

1.3.3 Створення документації за допомогою сторонніх утиліт

Цей метод передбачає використання сторонніх кросплатформенних програм для роботи з API. Найчастіше для цього використовується Postman – застосунок, який надає графічний інтерфейс для роботи з HTTP-протоколом, дозволяє експортувати та імпортувати запити в колекції, додавати опис та коментарі до кожного запиту та навіть писати тести за допомогою JavaScript [18]. Postman вперше здобув популярність у 2012 році і став надійною та нативною для найпопулярніших операційних систем програмою, яку зараз використовують понад 8 мільйонів розробників та 300 000 компаній [19]. Інші програми мають доволі обмежений набір функціональних можливостей у порівнянні з Postman.

Даний метод забезпечує актуальність документації за умови, що на кожен запит в Postman написаний тест, але не дозволяє гарантувати наявність нових ендпоінтів в документації. Але завдяки багатофункціональному користувацькому інтерфейсу, навіть якщо тести не використовуються, існує можливість легко перевірити, чи відповідає документація роботі програми та швидко виправити невідповідність.

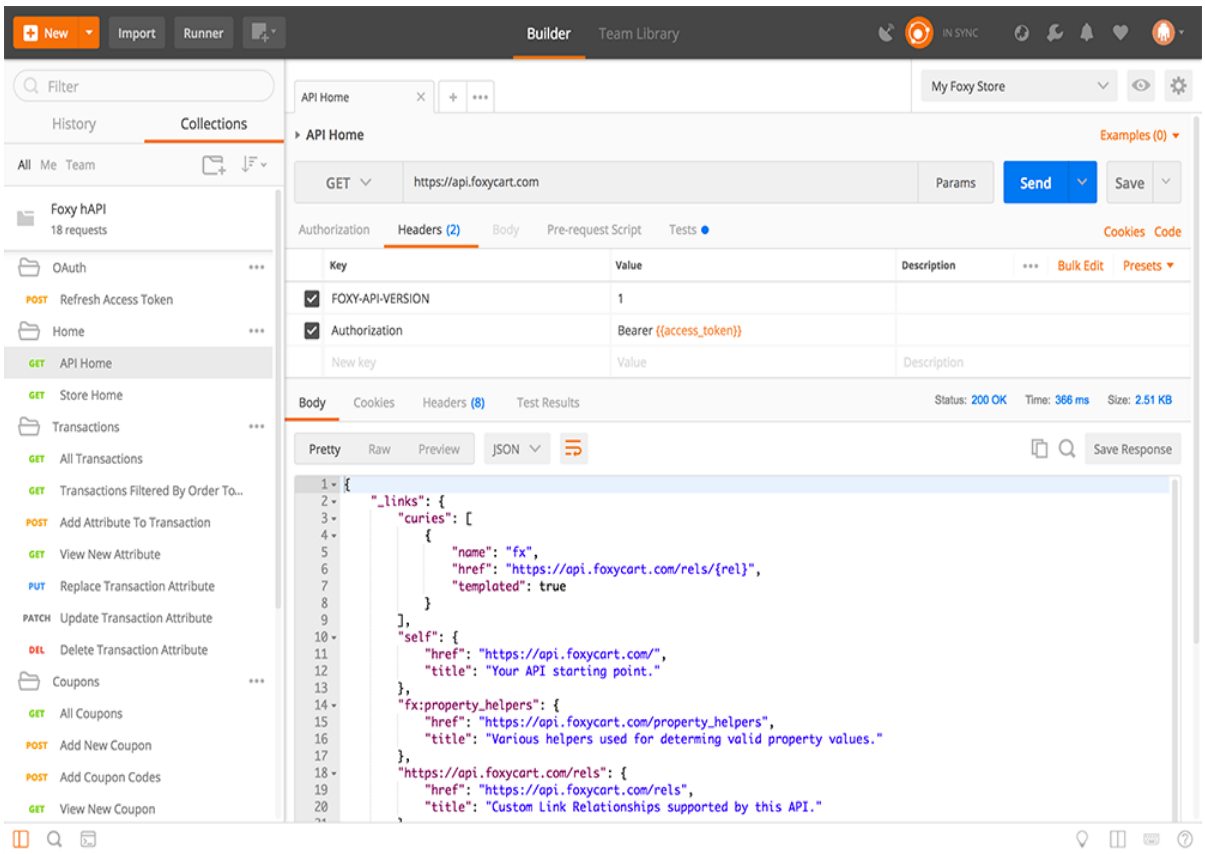


Рис. 1.6. Користувацький інтерфейс Postman

Як сказано вище, користувацький інтерфейс – головна перевага методу. Postman надає додаткові функціональні можливості на кшталт шаблонізації, що дозволяє легко перемикатись між локальним та розробницьким середовищем. Також до таких можливостей можна віднести швидкий пошук, автоматичне форматування для різних форматів даних, збереження історії.

Щодо часових затрат, то створення документації за допомогою цього методу все ж їх потребує, хоч і не так багато.

Додатково варто відзначити, що безкоштовні версії таких програм обмежені, що однозначно є одним з найголовніших недоліків. Необхідно, щоб всі члени команди встановили собі цю сторонню програму та навчились працювати з нею, що потребує часу. Також як і в методі написання документації вручну з’являється проблема з поширенням документації між членами команди, адже колекції, в яких зберігається документація, не знаходяться в системі контролю версій.

1.3.4. Метод на основі тестів

Метод на основі тестів базується на генерації документації під час запуску функціональних тестів. Його результатом є markdown-файл або набір із файлів формату HTML та JS. За допомогою системи неперервної інтеграції нескладно налаштувати автоматичне завантаження цих файлів на сервер, що дасть змогу кожному з членів команди легко переглядати документації у браузері.

Головна перевага цього методу полягає в тому, що він майже повністю забезпечує актуальність документації. Адже з додаванням тестів автоматично вносяться зміни в документацію. За допомогою встановлення високого мінімального відсотку покриття коду тестами можна мінімізувати «людський фактор» та забути про проблеми з невідповідністю документації реальному стану речей при динамічному розробленні програмного продукту.

Файл, який є результатом виконання тестів, визначає, чи буде графічний інтерфейс інтерактивним. Якщо це буде swagger-файл, то можна встановити будь-який графічний клієнт, який вміє працювати зі стандартом OpenAPI. В іншому випадку не буде можливості виконувати запити до API за допомогою документації. Бібліотека SpringRestDocs, яка є реалізацією даного методу, генерує документацію у своєму власному форматі не надає користувачам інтерактивності, що є її суттєвим недоліком.

Так як документація будується на основі тестів, даний метод вимагає часових затрат на їх написання. Водночас, зазвичай, точки доступу до API і так покриваються функціональними тестами, і при таких умовах цей метод не потребує додаткового часу на створення та підтримку документації.

1.4. Висновки

Вибір методу для створення документації для REST API в порівнянні з такими стандартами як GraphQL, JSON-RPC та SOAP відіграє важливу

роль у процесі розроблення мережевих програмних прикладних інтерфейсів, а задача оптимізації та автоматизації процесу створення документації для REST API є найбільш актуальною в порівнянні з іншими стандартами.

Важливі критерії для документації – це забезпечення її актуальності зі зведенням до мінімуму «людського фактору», її візуалізація та інтерактивність з використанням бібліотек для генерування графічного інтерфейсу за OpenAPI-файлом та мінімальні часові затрати на її створення та підтримку.

У табл. 1.1 наведено результати аналізу чотирьох методів для створення документації.

Таблиця 1.1

Порівняння методів для створення документації

Критерій/метод	створення документації за допомогою інструментів для конкретної мови програмування або фреймворку	створення документа-ції вручну	створення документації за допомогою сторонніх утиліт	створення документа-ції на основі тестів
забезпечує актуальність документації	–/+	–	–	+
надає графічний інтерфейс	+/–	–	+	+

Продовження табл.1.1

не потребує суттєвих часових	+	–	+/–	–/+

затрат				

Як бачимо, найбільшій кількості вимог задовольняють методи на основі тестів та з використанням інструментів для конкретних платформ. Натомість, якщо веб-сервіс покритий функціональними тестами, то часові затрати на документацію з використанням методу на основі тестів інколи навіть ефективніші за створення документації за допомогою інструментів для конкретної мови програмування/фреймворку. Методи створення документації вручну та за допомогою сторонніх утиліт задовольняють найменшій кількості вимог.

Наявна реалізація методу бібліотекою SpringRestDocs не дозволяє генерувати інтерактивний графічний інтерфейс, тому існує необхідність у створенні вдосконаленого методу, який би вирішив цю проблему.

2. МЕТОД СТВОРЕННЯ ДОКУМЕНТАЦІЇ НА ОСНОВІ ТЕСТІВ

2.1. Аналіз наявної реалізації методу створення документації на основі тестів

Наразі метод генерації документації для REST API на основі тестів реалізований у бібліотеці SpringRestDocs, яка є частиною екосистеми Spring. Принцип роботи бібліотеки зображений на рис. 2.1.

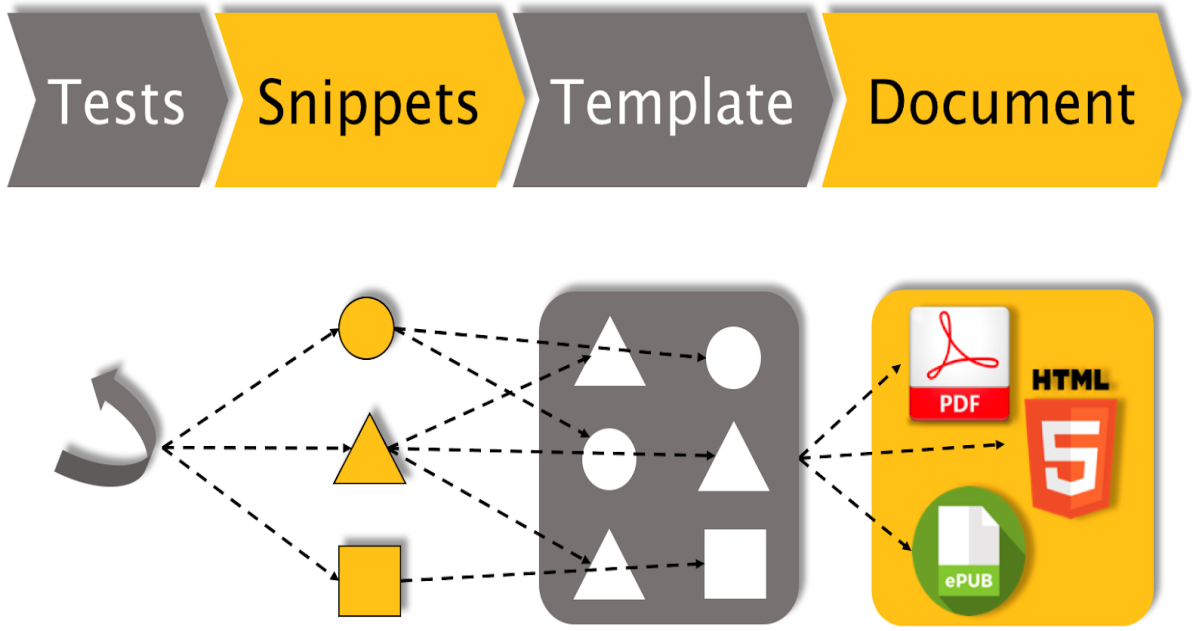


Рис. 2.1. Принцип роботи SprintRestDocs

Всі дії зі схеми відбуваються під час запуску тестів, окрім логіки верифікації ресурсів. Також відбувається генерація сніпетів. Сніпети представляють собою серіалізоване значення певного HTTP-атрибута, з яким взаємодіяв наш контролер. Ми готуємо спеціальний файл-шаблон, в якому вказуємо, в які секції повинні включатися згенеровані сніпети. На виході ми отримуємо скомпільований файл документації, також ми можемо задавати формат документації – це може бути html, pdf, epub, abook [20].

Наприклад, нехай ми маємо контролер, наведений на лістингу 2.1.

Лістинг 2.1. Приклад контролеру

```
@RestController
@RequestMapping(«/movies»)
public class MoviesController {

    @Autowired
    private MoviesRepository movieRepository;

    @GetMapping(path = «/{id}»)
    public ResponseEntity<MovieResource> getMovie(@PathVariable long
id) {
        return movieRepository.findOne(id)
                .map(movie -> ResponseEntity.ok(new
MovieResource(movie)))
                .orElse(new ResponseEntity(HttpStatus.NOT_FOUND));
    }
```

За допомогою класу SpringDataRepository ми звертаємось в базу даних за записом з ідентифікатором, який був переданий в контролер. SpringDataRepository повертає Optional – в разі, якщо в ньому є параметр ми виконуємо трансформацію JPA Entity в ресурс (при цьому ми можемо інкапсулювати поля, які ми не хочемо показувати у відповіді), якщо ж виконується умова Optional.isEmpty(), то ми повертаємо 404 код NOT_FOUND .

Базовий тест для цього контролера виглядає так як на лістингу 1.1.

Лістинг 2.2. Приклад тесту для контролера із лістингу 1.1

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
@AutoConfigureRestDocs(outputDir = «build/generated-snippets»)

public class SpControllerTest {

    @Autowired
    private MockMvc mockMvc;


    @Autowired
    private MovieRepository movieRepository;


    @After
    public void tearDown() {
        movieRepository.deleteAll();
    }


    @Test
    public void testGetMovie() throws Exception {
        // Given
```

```
            Movie                           movie                           =
Movie.builder().name(«Titanik»).genre(«Drama»).build();
            movieRepository.save(movie);
            // When
            ResultActions resultActions = s;
        }
    }
```

У тесті ми підключаємо автоконфігурацію mockMVC, RestDocs. Для RestDocs необхідно обов'язково вказати директорію, куди будуть генеруватись сніпети (в прикладі це «build/generated-snippets»). Це звичайний тест з використанням mockMvc, які практично щодня пишуть програмісти, які використовують Spring для тестування REST-сервісів. В даному прикладі використовується залежність spring.tests mockMvc, проте якщо використовується RestAssured, то приклад теж буде актуальним – є лише невеликі модифікації. Даний тест виконує виклик HTTP методу контроллера, робить верифікацію статусу, полів і роздруковує в консоль request / response flow.

За допомогою спеціального інтерфейсу ResultHandler під час запуску тестів ми побачимо тіло відповіді та запиту в консолі. Створивши свою реалізацією і підключивши її в тесті, ми можемо мати доступ до HttpRequest / HttpResponse, який виконувався в тесті і інтерпретувати результати виконання, при бажанні робити запис цих значень в консоль, зберігати в файловій системі, у власний файл документації та інше. Саме цю ідею з обробкою і реєстрацією використали автори бібліотеки SpringRestDocs. Тепер щоб згенерувати документацію,в тест за допомогою анотації потрібно підключити хендлер з класу MockMvcRestDocumentation. Після запуску тестів з доданою анотацією в в директорії «build/generated-snippets» буде створена папка з файлами-сніпетами. Сніпет– це серіалізована в файлі в текстовому поданні частина складової «HTTP request/response» корисного навантаження. Найбільш часто використовувані сніпети – curl-request, http-request, http-response, request-body, response-body, links, path-parameters, response-fields, headers.

Тепер необхідно підготувати файл шаблону і розмітити його в секції, куди будуть включатися згенеровані блоки сніпетів. Шаблон ведеться в форматі asciidoc, за замовчуванням він повинен знаходиться в директорії «src/docs/asciidocs». Приклад шаблону наведений на лістингу 2.3.

Лістинг 2.3. Приклад файлу-шаблону

```
== Rest convention
include::etc/rest_conv.adoc[]
== Endpoints
=== Movie
==== Get movie by ID
===== Curl example
include::{snippets}/sp-controller-test/test-get-movie/curl-
request.adoc[]
====== Success HTTP responses
include::{snippets}/sp-controller-test/test-get-movie/http-
response.adoc[]
====== Response fields
include::{snippets}/sp-controller-test/test-get-movie/response-
fields.adoc[]
====== HATEOAS links
include::{snippets}/sp-controller-test/test-get-movie/links.adoc[]
```

Варто зазначити, що в файлі-шаблоні завжди використовується шаблонізатор ascii-doc.

Після цього потрібно налаштувати збірку проекту і базову конфігурацію ascii-doctor.

В Spring Boot можна скористатися однією з його цікавих властивостей – всі ресурси, які знаходяться в директорії «src/static» або «src/public» будуть доступні як статичний контент при зверненні з браузера. Це дає нам можливість налаштувати збірку проекта так, що після збірки документації вона буде копіюватись її в директорію «static/docs». Незалежно від того, де проект буде розгорнутий і на якому оточенні він буде знаходиться – користувачам API завжди буде доступна актуальна версія документації.

Всі вище описані дії з налаштування необхідно провести один раз при запуску проекту. Після цього залишається лише додавати нові тести та модифікувати шаблони.

Також бібліотека володіє методами, які дозволяють:

- вказувати вхідні GET-параметри та їх опис;
- вказувати вхідні параметри в тілі запиту та їх опис;
- додавати в шаблони посилання на інші методи та поля;
- вказувати вхідні та вихідні заголовки та їх опис;
- додавати інформацію про обмеження на поля моделей.

2.2. Недоліки наявної реалізації методу створення документації на основі тестів

Найперше, варто відзначити, що цей метод не повністю мінімізує людський фактор, адже потрібно кожен раз вносити зміни в шаблони, а про це можна легко забути.

Також процес налаштування SpringRestDocs доволі складний та громіздкий, але це можна пояснити складністю екосистеми Spring.

На мою думку, найголовнішим недоліком SpringRestDocs є відсутність інтерактивності в документації. Документація, згенерована бібліотекою зображена на рис. 2.2.

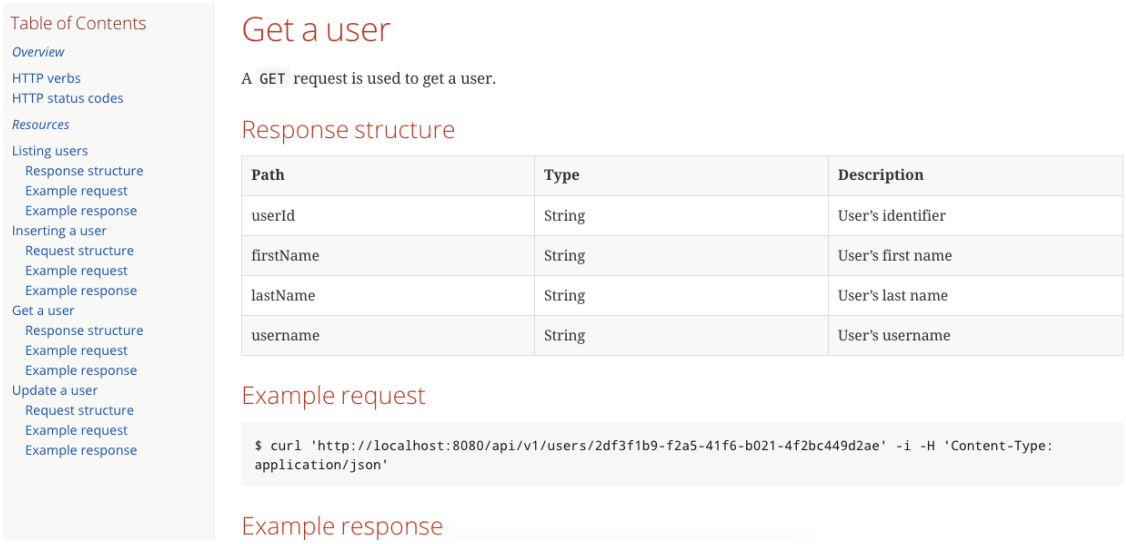


Рис. 2.2. Документація, згенерована в SpringRestDocs

Користувачі API хоч і матимуть змогу перейшовши за посиланням побачити список доступних ресурсів та методів, вони все одно будуть змушені користуватись сторонніми програмами для того, щоб почати

розробляти клієнтську частину. Слід також зазначити, що додатковою причиною для цього є те, що SpringRestDocs не вміє генерувати код клієнтської частини, лише сніпети у заданому форматі.

Наступним недоліком є відсутність можливості вказати принцип авторизації для API. На даний момент існує декілька способів авторизації в REST API. Зокрема, це BasicAuth, BearerAuth, ApiKeyAuth, OpenID та OAuth2. У випадку використання будь-якого з них в тестах доведеться дублювати запити для кожного з тестів. Адже заголовки потрібно вказувати для кожного ендпоінту окремо, як це показано на лістингу 2.4.

Лістинг 2.4. Додавання заголовків в документацію

```
this.mockMvc
        .perform(get(«/movies»).header(«Authorization», «Basic
dXNlcjpzZWNyZXQ=»))
        .andDo(document(«headers»,
                requestHeaders(headerWithName(«Authorization»).description(
                                «Auth credentials»)),
                responseHeaders( headerWithName(«X-RateLimit-
Limit»).description(
                                «Total number of requests allowed»),
                headerWithName(«X-RateLimit-Remaining»).description(
                                «Remaining count of requests for current
period»),
                        headerWithName(«X-RateLimit-Reset»).description(
                                «Time when the current period will
reset»))))
```

2.3. Запропонований метод створення документації на основі тестів

Фактично, всі ці недоліки усуваються використанням стандарту OpenAPI. Специфікація OpenAPI, початково відома як Swagger – це специфікація машиночитабельних файлів з інтерфейсами для опису, створення, використання і візуалізації REST веб-сервісів [21]. Існує багато різних інструментів, що дозволяють генерувати документацію, код та тести за файлом з описом інтерфейсу. За розробкою специфікації OpenAPI (OAS) наглядає Open API Initiative, проект Linux Foundation. Розробка Swagger почалась в 2010 році. Специфікація OpenAPI не залежить від мови програмування. Також її можна поширювати на нові технології і не тільки HTTP-протоколи. З декларативною специфікацією ресурсу OpenAPI,

клієнти можуть розуміти і використовувати сервіси без знання деталей реалізації сервера.

Принцип використання цієї специфікації у даному методі полягає в тому, що замість використання сніпетів, шаблонів та кінцевих файлів потрібно спочатку генерувати машиночитабельний Swagger-файл, а потім вже цей файл перетворювати у людиночитабельну документацію. Результатом цього може бути файл у форматі PDF, HTML тощо.

Кінцевий HTML-файл містить javascript-код, який дозволяє авторизовуватись в API та відсилати запити на сервер прямо в браузері. Його вигляд для користувача у браузері зображений на рис. 2.3.

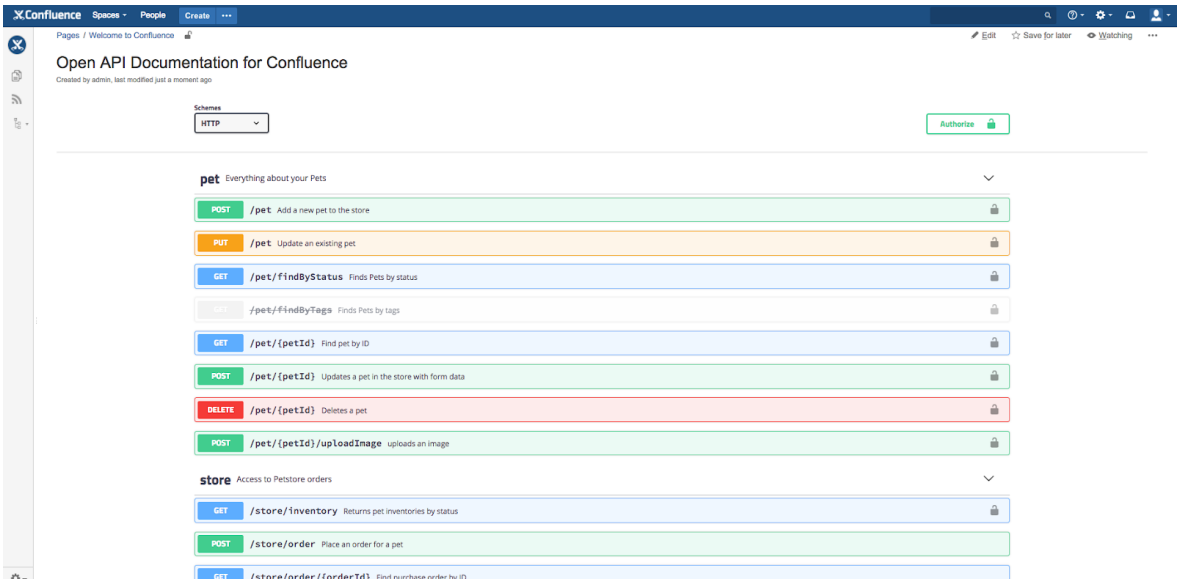


Рис. 2.3. Документація з використанням OpenAPI

Цей спосіб не потребує ручного редагування шаблонів, але водночас змінити даний шаблон буде неможливо.

Також специфікація OpenAPI має підтримку всіх найпоширеніших способів авторизації. Для цього потрібно лише додати у Swagger-файл секцію securitySchemes. Наприклад, для ApiKey Auth це буде виглядати так:

Лістинг 2.5. Конфігурація для авторизації способом ApiKey Auth

```
components:
  securitySchemes:
    BasicAuth:
```

```
    type: http
    scheme: basic

  ApiKeyAuth:
    type: apiKey
    in: header
    name: X-API-Key
```

При цьому наш метод не повинен відповідати за графічну візуалізацію документації. Результатом його роботи є Swagger-файл. Програмісти будуть самі обирати спосіб візуалізації, та те, коли буде генеруватись Swagger-файл, адже це можна зробити декількома способами. Перший складається з двох етапів, зображених на рис. 2.4.

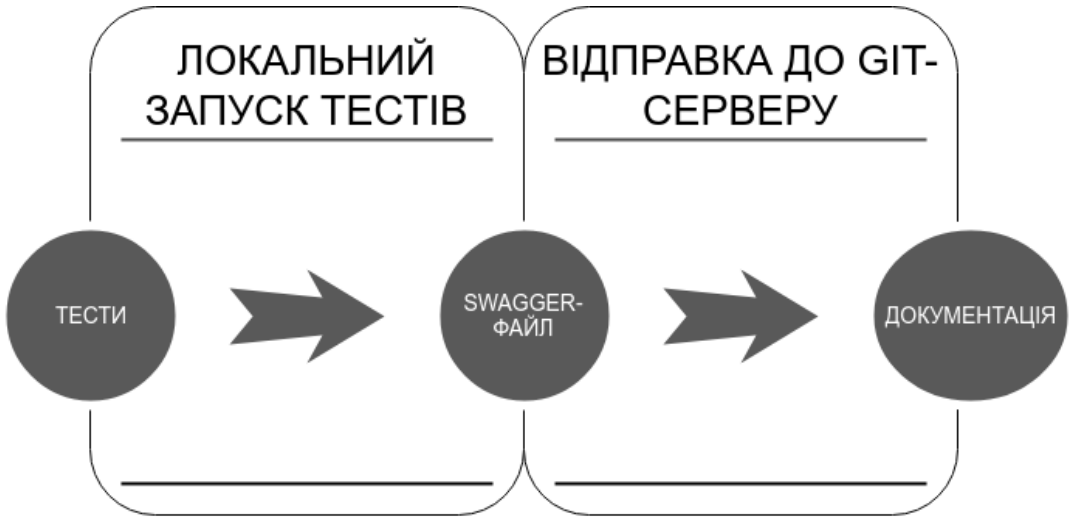


Рис. 2.4. Спосіб №1 генерації Swagger-файлу та його візуалізації

Цей спосіб найпростіший. На першому його етапі локально на ПК програміста запускаються тести, після чого в разі успіху у директорії, яка вказана у конфігурації, генерується swagger-файл. Цей файл повинен бути статичним – тобто таким, що доступний на сервері для скачування ззовні за посиланням. Після додавання змін в систему контролю версій та вивантаження змін на сервер, цей файл буде доступний за посиланням. Візуалізувати документацію можна, використавши це посилання за допомогою бібліотеки swagger-ui, підключивши її до проекту або хмарних інструментів для візуалізації на кшталт SwaggerHub.

Відмінність другого способу полягає в тому, що тести запускаються не локально, а в CI – системі неперервної інтеграції, що зображено на рис. 2.5.

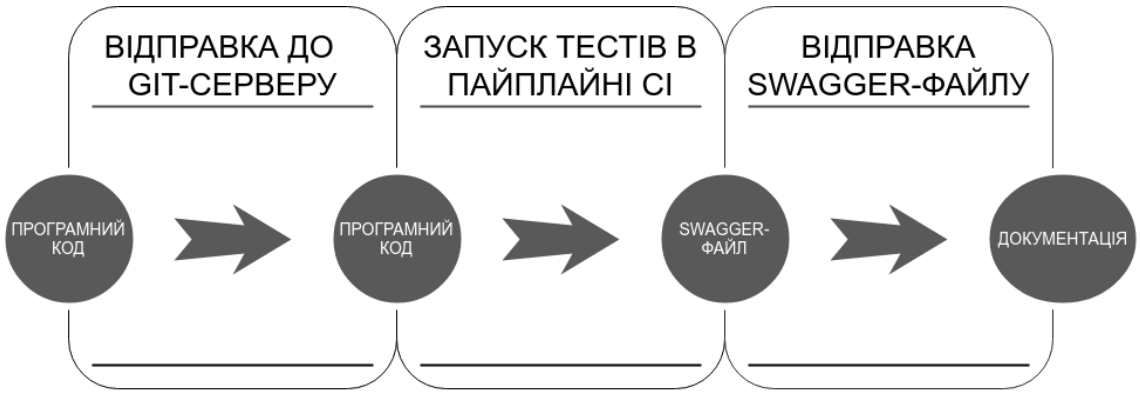


Рис. 2.5 Спосіб №1 генерації Swagger-файлу та його візуалізації

У такому випадку в системі неперервної інтеграції ми маємо можливість перевірити відсоток покриття тестами. Після вдалого запуску тестів за допомогою тієї ж системи неперервної інтеграції файл відправляється на сервер, який зберігає такі файли та візуалізує їх. Роль сервера може грати і сам проект. В випадку мікросервісної архітектури ми можемо створити окремий сервіс, який буде це робити.

2.4. Висновки

Наразі метод генерації документації для REST API реалізований у бібліотеці SpringRestDocs, яка є частиною екосистеми Spring. Принцип роботи цього методу полягає у додаванні нового обробника, який реалізує спеціальний інтерфейс до процесу обробки генерації сніпетів, з яких пізніше створюються шаблони, а потім генерується результат у вигляді файлу в форматі ePub, PDF або HTML в залежності від конфігурації. В бібліотеці реалізовано багато функціональних можливостей, таких як додавання посилань, додавання опису для вхідних та вихідних параметрів, вхідних заголовків тощо.

Але поточна реалізація має ряд суттєвих недоліків. Найбільший недолік полягає у тому, що кінцева документація є статичною, а не інтерактивною. Також відсутні механізми для того, щоб вказати спосіб авторизації, що призводить до дублювання коду в тестах. Також програміст повинен ще й вносити зміни в шаблони і ознайомитись зі стандартом ascii-doc, що потребує затрат часу і збільшує «людський фактор».

Всі ці недоліки усуваються новим принципом роботи методу. Зокрема, замість генерації сніпетів буде генеруватись Swagger-файл, який відповідає специфікації OpenAPI, в якій дані недоліки враховані та усунені.

3. ПРОГРАМНА РЕАЛІЗАЦІЯ МЕТОДУ

3.1. Обґрунтування вибору технологій

3.1.1. Обґрунтування вибору мови програмування

Згідно статистики [22], найпопулярніші мови програмування для серверних частин веб-сайтів – PHP, C#, Ruby.

PHP – мова програмування, призначена для створення веб-застосунків. Була створена у 1995 році Расмусом Лердорфом. На даний момент, близько 80% веб-сайтів у світі написані на PHP [23]. Такій великій кількості сприяли сайти, створені за допомогою CMS-систем – програмного забезпечення, що призначене для створення веб-сайтів чи інших інформаційних ресурсів в Інтернеті чи окремих комп'ютерних мережах [24].

PHP може працювати у трьох режимах – CGI, Fast CGI та FPM [25]. Common Gateway Interface – це стандарт, який описує, як веб-сервер повинен використовувати програми/скрипти, як потрібно передати їх параметри HTTP-запиту, як програми повертають результати своєї роботи веб-серверу. Якщо коротко, то його суть полягає в тому, що на кожен HTTP-сервер веб-сервер запускає новий процес. FastCGI – продуктивніша та безпеченіша версія технології CGI. Проблема CGI-програм у тому, що вони перезапускаються веб-сервером при кожному запиті, що призводить до зниження продуктивності. FastCGI усуває цю проблему, роблячи процес постійно запущеним і передаючи йому HTTP-запити на обробку. FastCGI Process Manager (менеджер процесів FastCGI) – це альтернативна реалізація FastCGI режиму в PHP з декількома додатковими можливостями, які зазвичай використовуються для високонавантажених сайтів.

На мою думку, один з найбільших плюсів PHP – гнучкість та універсальність. Зокрема, на цій мові програмування можна писати як в процедурному, так і в об’єктно-орієнтованому стилі. Вона чудово

підходить для новачків, оскільки має доволі низький поріг виходу, але водночас на ній можна писати масштабні та складні проекти. PHP працює доволі швидко, якщо порівнювати, наприклад, з Python та потребує небагато системних ресурсів.

Оскільки PHP має відкритий сирцевий код, то його підтримують багато розробників, і помилки в роботі досить швидко знаходять і виправляють, що робить PHP стабільним програмним забезпеченням. Має розвинену екосистему, можна легко знайти функціональні модулі та бібліотеки для роботи з PDF, графіками, різними СУБД тощо.

Як недоліки, можна виділити синтаксис, який багато людей вважають незручним. Також до мінусів традиційно відносять слабку типізацію, однак в сьомій, останній версії мови, є багато нових функціональних можливостей, які дають змогу застосовувати строгу типізацію.

C# – об'єктно-орієнтована мова програмування від Microsoft, яка поєднує обчислювальну потужність C ++ з простотою Visual Basic. В основу C# ліг C++, також ця мова містить функції, подібні до функцій Java. C# в основному призначений для роботи з платформою .Net Microsoft [26].

C# можна використовувати для створення майже будь-чого, але особливо ця мова використовується у створенні настільних застосунків та ігор для Windows. C# також може використовуватися для розробки веб-застосунків і стає все більш популярною і для мобільних розробок. Для створення веб-сайтів частіше всього використовують ASP.NET.

ASP.NET – це фреймворк, призначений для створення динамічних веб-сторінок [27]. Проста веб-сторінка HTML є статичною; її вміст визначається фіксованою розміткою HTML, яка знаходиться на сторінці. Динамічні сторінки, як і ті, які створюють за допомогою веб-сторінок ASP.NET, дозволяють створювати вміст сторінки на льоту, використовуючи код, написаний на мові C#.

До переваг даного стеку можна віднести MVC-архітектуру, багатофункціональність, простоту та гнучкість. Також постійний моніторинг – потужна особливість ASP.NET. Вам не доведеться турбуватися про стан програм, компонентів та самих сторінок. Програма спостерігає за будь-якими недозволеними подіями , і, якщо щось трапиться (наприклад, працюватиме небажаний нескінченний цикл), то ASP.NET перезапуститься самостійно.

Створення веб-застосунків за допомогою C# – доволі дорога справа, оскільки потрібні витрати на ліцензії для Visual Studio. Також документація фреймворку не настільки хороша, як хотілося б, і під час створення програм MVC ви можете зіткнутися з проблемами, які не описані в документації. Також даний стек має проблеми з оберненою сумісністю.

Ruby – це інтерпретована, об'єктно-орієнтована мова програмування з чіткою динамічною типізацією. Ruby розглядається як гнучка мова, оскільки дозволяє користувачам вільно змінювати свої складові частини. Основні частини Ruby можна за власним бажанням видалити або переробити. Існуючі деталі можна додати. Тобто мова Ruby не обмежує дії програміста [28].

Для створення веб-застосунків на Ruby частіше всього використовують Ruby on Rails – фреймворк, який дає веб-розробникам структурну основу для всього коду, який вони пишуть. Він дозволяє програмістам створювати веб-сайти та програми, абстрагуючись та спрощуючи більшість завдань, що повторюються.

Переваги Ruby – елегантний синтаксис, простота та швидкість в розробці. Ще одна чудова риса – активна спільнота розробників, яка активно пише докладні технічні посібники та створює уроки. Вони також проводять різні конференції та зустрічі, на яких програмісти із задоволенням діляться своїм досвідом та допомагають знайти найкраще рішення для проекту.

До недоліків можна віднести кількість людей, які володіють даним стеком технологій. Кількість розробників на Ruby постійно збільшується, але ця кількість все ще суттєво відстає від PHP та C#. Також добре відомим є факт, що Ruby не може похвалитися високою швидкістю виконання. Однак це не все так погано, оскільки версія 2.6.3, випущена у квітні 2019 року, демонструє значне поліпшення продуктивності порівняно з попередніми версіями.

Задача розроблення методу створення документації для REST API актуальна для всіх трьох мов програмування, однак було обрано мову програмування PHP, зважаючи на її поширеність, гнучкість та простоту.

3.1.2. Обґрунтування вибору фреймворку

Попри те, що в системі PHP існує PSR – набір стандартів, які створені для того, щоб сумістити інтерфейси різних фреймворків, не вийде створити бібліотеку, яка буде повністю універсальною для всіх фреймворків. Саме тому наша задача – винести логіку, яка не залежить від фреймворку, в спільний простір імен та обрати фреймворк, в якому буде працювати бібліотека. В даний момент в світі PHP існують два найпоширеніших фреймворки – Symfony та Laravel.

Laravel – один з найпопулярніших веб-фреймворків PHP, які слідують патерну MVC. Створений Тейлором Отвеллом, даний фреймворк є безкоштовним та має відкритий сирцевий код [29]. Принципи, які сповідує Laravel – менше коду та менше часу. Він призначений в першу чергу для того, щоб ефективно використовувати час програміста. Перший реліз фреймворку був у 2011 році, зараз мінорні релізи проходять кожні півроку. У 2015 році за результатами сайту sitepoint.com серд PHP-фреймворків за результатами опитування програмістів зайняв перше місце у номінаціях: «Фреймворк корпоративного рівня» та «Фреймворк для особистих проектів».

Багато функціональності Laravel надає «з коробки». Зокрема, це – аутентифікація, механізм черг, кешування, міграції бази даних, роутинг та

інші. Laravel легко розширюється. Необхідні модулі для фреймворка підключаються у вигляді пакетів-провайдерів. У версії Laravel 5.5 достатньо просто встановити пакет через Composer, і він відразу буде доступний без необхідності що-небудь писати в коді.

Окремо варто відзначити Eloquent ORM – красиву і просту реалізацію патерну Active Record в Laravel для роботи з базами даних. Кожна таблиця має відповідний клас-модель, який використовується для роботи з цією таблицею. Функціональність Eloquent ORM дозволяє повністю убезпечити програмне забезпечення від атак типу SQL Injection, а також завантажувати дані з декількох таблиць (вирішуючи проблему N+1) або ж обробляти дані з бази даних частинами.

Також Laravel постачає функціональні можливості для написання модульних та інтеграційних тестів. Незважаючи на це, фреймворком в основному користуються, коли важливий критерій часу, тому тести в проектах на Laravel пишуться не так часто.

Варто зазначити, що Laravel побудований на компонентах Symfony – більш низькорівневого та складного фреймворку. Symfony – це не просто фреймворк, а ціла екосистема. Він складається з багатьох незалежних компонентів, які можна використовувати окремо, та на основі яких створено багато інших фреймворків та CMS-систем. Зокрема, ці компоненти використовуються такими технологіями, як Sylius eCommerce, Delicious, API Platform, Oro, Magento, Drupal, Opencart. Отже, як сказано вище, Symfony – це набір компонентів, які можна перевикористати та MVC-фреймворк одночасно.

Сам фреймворк не містить компонентів для роботи з базою даних, але він тісно інтегрований з бібліотекою Doctrine. Doctrine – це PHP-бібліотека, яка є реалізацією патерну Data Mapper та представляє зручні методи для управління базою даних через об’єктно-орієнтоване програмування. Для роботи з реляційними БД Doctrine має компонент, який називається ORM. За допомогою цього компоненту таблиці в базі

даних співставляється PHP-клас (з точки зору DDD він також називається классом-сутністю).

Symfony реалізує функціональність поштової програми на основі популярної бібліотеки Swift Mailer. Ця поштова програма підтримує відправку повідомлень з поштових серверів, а також використання популярних поштових провайдерів, таких як Mandrill, SendGrid і Amazon SES [30].

Механізм інтернаціоналізації дозволяє встановити і здійснити переказ повідомлень веб-додатки на основі обраної мови або країни.

Symfony пропонує систему логування помилок веб-застосунку за допомогою бібліотеки Monolog.

Отже, даний фреймворк має потужну екосистему з хорошою спільнотою та багатьма розробниками. Документація має дуже високу якість та постійно оновлюється для всіх версій. Також фреймворк пропонує механізм функціональних і модульних тестів для знаходження помилок в веб-застосунку. Symfony чудово підходить для складних і навантажених веб-проектів та електронної комерції.

До мінусів можна віднести те, що попри хорошу документацію, фреймворк є складним для вивчення.

Саме на Symfony буде розроблятись запропонований метод створення документації для REST API на основі тестів.

3.2. Інтерфейс, який надає програмне забезпечення

Бібліотека Symfony Rest Docs надає інтерфейс у вигляді доступного для використання класу Request, який знаходиться в кореневому просторі імен. Також бібліотека надає можливість вказувати вихідні параметри для Swagger-файлу через конфігурацію. Для цього потрібно в Symfony-проекті в папці Symfony створити файл rest_docs.yaml. Варто зазначити, що без цього файлу запуск тестів завершиться помилкою, адже для Swagger-файлу згідно специфікації OpenApi [31] потрібно задати обов’язкові

параметри «info.name» та «info.version». Всі параметри, які можливо вказати в конфігурації перелічені в табл. 3.1.

Таблиця 3.1

Можливі параметри конфігурації

Ключ параметру	Опис
output	Шлях до swagger-файлу, який генеруватиметься під час запуску інтеграційних тестів
swagger	Об’єкт, який містить конфігурацію для swagger-файлу.
swagger.info	Об’єкт, який містить метадані про API. Метадані можуть використовуватись клієнтами за потреби.
swagger.info.title	Рядок. Назва застосунку. Обов’язковий параметр.
swagger.info.description	Рядок. Короткий опис застосунку. За потреби може бути markdown-синтаксис.
swagger.info.termsOfService	Рядок. Правила використання API.
swagger.info.contact	Об’єкт. Контактна інформація для користувачів API.
swagger.info.contact.name	Рядок. Ім’я контактної особи або організації.
swagger.info.contact.url	Рядок. Посилання на веб-сайт контактної особи або організації.
swagger.info.contact.email	Рядок. Електронна пошта контактної особи або організації.
swagger.info.license	Об’єкт. Інформація про ліцензію.
swagger.info.license.name	Рядок. Назва ліцензії.
swagger.info.license.url	Рядок. Посилання на ліцензію, яка використовується API. Повинен бути у форматі URL.

Продовження табл.3.1

swagger.info.version	Рядок. Версія API. Обов’язковий параметр.
swagger.host	Рядок. Хост (доменне ім’я або IP-адреса). Повинен містити тільки хост – без схеми або шляхів. Може містити порт.
swagger.schemes	Масив рядків. Список протоколів, які підтримує API. Допустимі значення: «http», «https», «wss» .
swagger.consumes	Масив рядків. Список MIME-типів, що API може приймати.
swagger.produces	Масив рядків. Список MIME-типів, що API може повертати.
swagger.definitions	Масив об’єктів Містить типи даних, які можуть використовуватись у запитах чи відповідях.
swagger.parameters	Масив об’єктів. Містить список вхідних параметрів, які можуть бути перевикористані в запитах.
swagger.responses	Масив об’єктів. Містить список відповідей, які можуть бути перевикористані в запитах.
swagger.securityDefinitions	Масив об’єктів. Опис схем авторизації, які можна перевикористати в запитах.
swagger.security	Масив рядків. Список схем авторизації, які використовуються у всіх запитах.
swagger.tags	Масив об’єктів. Список тегів, використовуваних специфікацією, з додатковими метаданими. Порядок тегів може бути використаний для відображення їх порядку за допомогою інструментів розбору.
swagger.externalDocs	Об’єкт. Інформація про додаткову документацію, якщо така існує.

Отже, через конфігурації можна задати всі параметри для вихідного Swagger-файлу, окрім «paths». Клас Request під час виклику методу send(), який приймає як параметр об’єкт типу KernelBrowser, зчитає під час першого виклику – з конфігурації, а під час наступних – з проміжного файлу список операцій, та дописує в файл нову операцію. Для операції в тесті можна вказати багато параметрів за допомогою наступних методів:

- withTags(array $tags) – дозволяє задати список тегів для операції. Теги використовуються для логічного групування операцій за ресурсами. Тобто, в параметр передається список назв секцій, в яких буде розміщена операція у вихідній документації;
- withSummary(string $summary) – призначений для того, щоб задати короткий підсумок того, що робить операція. Для максимальної читабельності в swagger-ui це поле повинно бути менше 120 символів;
- withDescription(string $description) – надає можливість докладного опису та пояснення поведінки операції. Може бути використаний markdown-синтаксис;
- withExternalDocs(string $url, ?string $description) – дозволяє задати додаткову зовнішню документацію для операції;
- withOperationId(string $operationId) – дозволяє вказати унікальний рядок, яка має назву operationId і використовується для ідентифікації операції. OperationId повинен бути унікальним серед усіх операцій, описаних в API;
- consumes(array $consumes) або produces($produces) – надає можливість вказати список MIME-типів, які може приймати або, відповідно, повертати ця операція. Виклик даної функції перезаписує значення з глобальної конфігурації для операції. Пусте значення може бути використане для очищення глобального визначення;
- withHeader(string $name, array $options, $value) – додає до операції

та запиту заголовок з ключем $name. Масив $options – асоціативний масив, в який можна передати ключі та значення, які описані в специфікації OpenApi 2.0 для параметрів. Зокрема, це ключ «type», який є обов'язковим;

- withPathParameter(string $name, array $options, $value) – дозволяє задати параметр в URL. Наприклад, якщо URL операціїї «tasks/{taskId}», то за допомогою цього методу можна вказати опції та значення для параметра taskId;
- withQueryParameter(string $name, array $options, $value) – надає можливість додати GET-параметр до операції;
- withBody(array $options, $value) – дозволяє описати та вказати тіло запиту;
- withParameter(string $reference, $value) – дозволяє додати до запиту та операції параметр з конфігурації, який описаний в секції «parameters»;
- withSecurity(array $requiredSecuritySchemes) – надає можливість застосувати схеми авторизації, описані в конфігурації, до конкретної операції;
- markAsDeprecated() – позначає операцію як «deprecated» – тобто, такою, що невдовзі буде видаленою.expectsResponse(int $responseCode, array $options) – найголовніший метод, який дозволяє вказати очікуваний код відповіді, та масив опцій, серед яких – «description», «headers», «example» та «schema». Опції описані детальніше в специфікацій OpenAPI, а наразі варто зазначити, що ключ «description» – обов'язковий.

Всі вище описані методи повертають об'єкт типу Request, що робить можливим виклик всіх методів ланцюжком, як це показано на лістингу 3.1.

Лістинг 3.1. Приклад використання класу Request

```
(new Request('GET', '/api/tasks'))
    ->withSummary('Get tasks list')
    ->withDescription('Get user tasks ordered in alphabetic order by
description')
```

```
    ->produces(['application/json'])
    ->consumes(['application/json'])
    ->withTags(['Tasks'])
    ->withHeader(
        'Authorization',
        [
            'type'        => 'string',
            'description' => 'Auth token. Example: `Bearer {token}`'
        ],
        'Bearer 123'
    )
    ->expectsResponse(200, [
        'description' => 'Returns list of tasks',
        'schema'      => [
            'type'       => 'object',
            'required'   => ['tasks'],
            'properties' => [
                'tasks' => [
                    'type'  => 'array',
                    'items' => [
                        'type'       => 'object',
                        'required'   => [
                            'id',
                            'title',
                            'description'
                        ],
                        'properties' => [
                            'id'          => [
                                'type' => 'integer'
                            ],
                            'title'       => [
                                'type' => 'string'
                            ],
                            'description' => [
                                'type' => 'string'
                            ]
                        ]
                    ],
                ]
            ]
        ]
    ])
```

Результатом виконання коду, який наведений на лістингу 3.1 є додавання операції до Swagger-файлу, що наведена на лістингу 3.2.

Лістинг 3.2. Результат виконання коду з лістингу 3.1

```
"paths": {
  "/api/tasks": {
    "get": {
      "responses": {
        "200": {
          "description": "Returns list of tasks"
        }
      },
      "tags": [
        "Tasks"
      ],
      "summary": "Get tasks list",
```

```
            "description": "Get user tasks ordered in alphabetic order by
description",
            "produces": [
              "application/json"
            ],
            "consumes": [
              "application/json"
            ],
            "parameters": [
              {
                "name": "Authorization",
                "in": "header",
                "type": "string",
                "description": "Auth token. Example: `Bearer {token}`"
              }
            ]
          }
        }
      }
```

3.3. Архітектура розробленого програмного забезпечення

Програмний метод реалізований у вигляді бібліотеки. Якщо точніше, то ця бібліотека встановлюється у проект як залежність через Composer.

Composer – менеджер пакетів прикладного рівня для PHP, який забезпечує стандартний формат для управління залежностями та необхідними бібліотеками в проекті [32]. Його розробили Ніл Адерман і Хорді Боггіано, які і досі підтримують цей проект [33]. Розробка почалася у квітні 2011 року і перший реліз відбувся вже у березні 2012. При розробці автори брали натхнення з «npm» для Node.js і «bundler» для Ruby [34].

Менеджер пакетів працює за допомогою командного рядка і встановлює залежності для застосунку. Також Composer дозволяє користувачам встановлювати PHP-пакети, доступні на packaglist.org, який є основним публічним сховищем, що зберігає пакети. Також він також реалізує автозавантаження класів для встановлених бібліотек і це полегшує використання коду від сторонніх розробників. Для того, щоб у будь-якому проекті була доступна ця бібліотека, на packaglist.org було зареєстровано пакет «sushchyk/symfony-rest-docs», де «sushchyk» – це назва вендору, тобто поставника, а «symfony-rest-docs» – назва самого пакету. Також був налаштований Web Hook із github.com на packaglist.org. Завдяки Web

Hook зміни, які відправлені на GitHub, відразу потрапляють на Packaglist, і можуть бути стягнуті у проект шляхом оновлення відповідної бібліотеки.

Вміст кореневої директорії бібліотеки зображений на рис. 3.1.

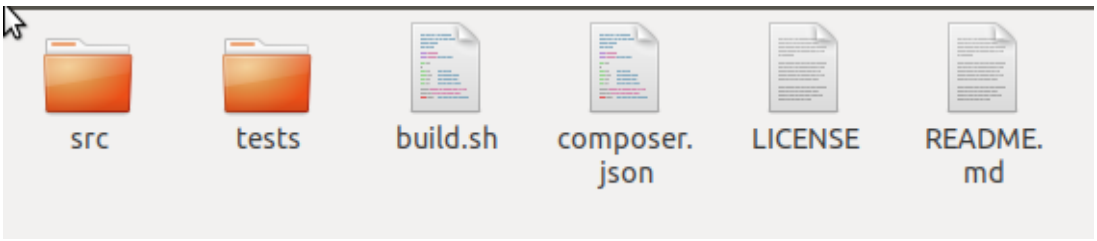


Рис. 3.1. Вміст кореневої директорії

В директорії «src» знаходяться безпосередньо код бібліотеки, який лежить у просторі імен «Sushchyk/SymfonyRestDocs». В директорії «tests» знаходяться автоматичні тести для проекту. Файл «build.sh» містить конфігурацію для системи неперервної інтеграції. На кожен коміт та пуш Travis CI, яка налаштована в Github-репозиторії з проектом, виконує скрипт «build.sh», який, в свою чергу, виконує збірку проекту та запускає автоматичні тести. Його вміст можна побачити на лістингу 3.3.

Лістинг 3.3. Вміст файлу build.sh

```
##!/usr/bin/env bash
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');" &&
php -r "if (hash_file('sha384', 'composer-setup.php') === 'a5c698ffe4b8e849a443b120cd5ba38043260d5c4023dbf93e1558871f1f07f58274fc6f4c93bcfd858c6bd0775cd8d1') { echo 'Installer verified'; } else { echo 'Installer corrupt'; unlink('composer-setup.php'); } echo PHP_EOL;" &&
php composer-setup.php &&
php -r "unlink('composer-setup.php');" &&
php composer.phar install &&
./vendor/bin/phpunit tests
```

В файлі «composer.json» міститься назва, опис, ліцензія, версія та тип пакету. Також там вказані його залежності. Зокрема, це:

- phpunit/phpunit – бібліотека для тестування;
- symfony/framework-bundle- – фреймворк Symfony;
- symfony/browser-kit – бандл Symfony, який використовується в тестах для HTTP-запитів.

Файли «LICENSE» та «README.MD», в свою чергу, містять умови ліцензії та інструкції щодо встановлення/використання бібліотеки.

Отже, основний код розміщений в директорії «src», якій відповідає простір імен «Sushchyk/SymfonyRestDocs». В свою чергу, в цій директорії файли розбиті за просторами імен за їхнім призначенням – «Exception», «OpenApi», «Symfony», «Utils».

В просторі імен «Exception» всього один клас – DataConflictsWithSchemaException, який наслідує стандартний PHP-клас Exception, що зображено на рис. 3.2.

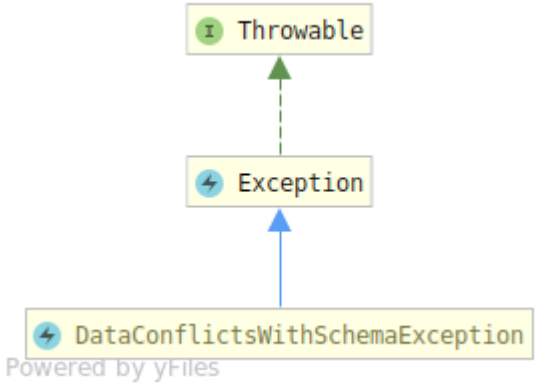


Рис. 3.2. Діаграма класів в просторі імен «Exception»

Це виключення кидається у випадку, якщо тіло відповіді не співпадає з очікуваною схемою, яку користувач передав під ключем «schema» в параметрі «options» методу «expectsResponse». В такому випадку кидається AssertionError з відповідним текстом помилки. Текст може містити наступні помилки, де замість «%data%» підставляється, наприклад, «item #0, property `title`»:

- expecting %data% to be array;
- expecting %data% to be object;
- %data% is not defined;
- expecting %data% to be integer;
- expecting %data% to be Boolean;
- expecting %data% to be string;

- unexpected key %data%;
- undocumented key %data%;
- header %data% not found in response.

Найбільше класів міститься у просторі імен «OpenApi», що зображено на рис. 3.3.

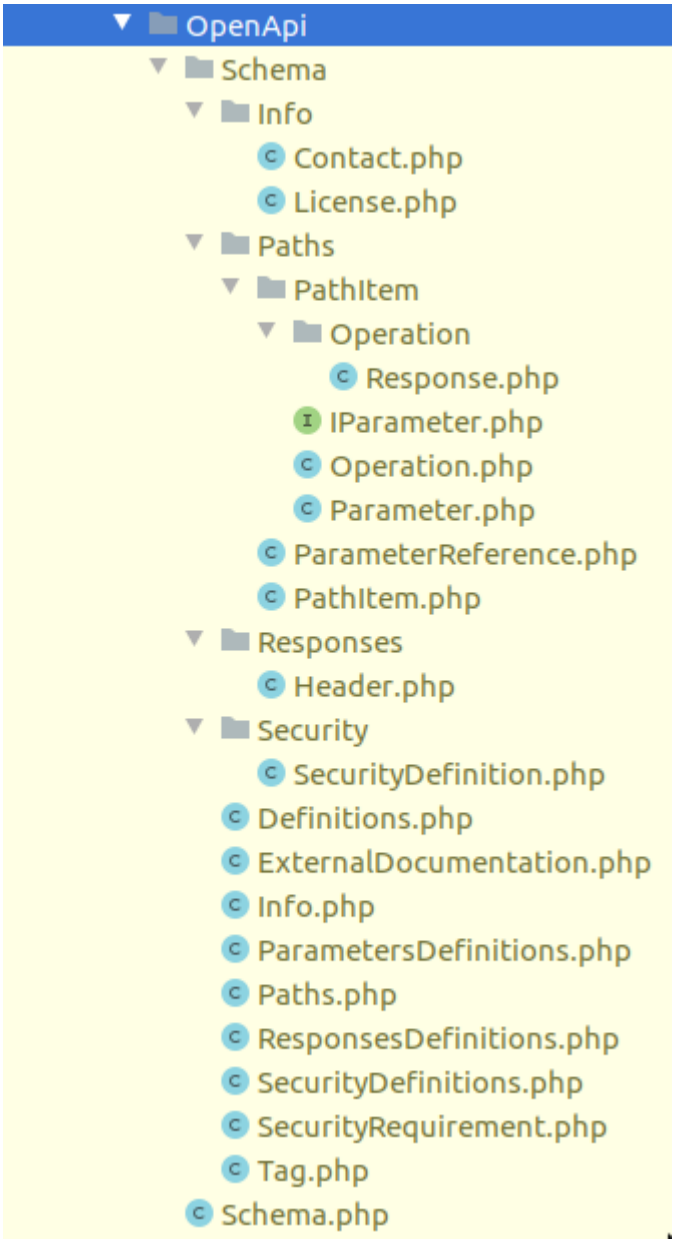


Рис. 3.3. Класи в просторі імен «OpenApi»

Усі класи в цьому просторі імен не містять жодної логіки, лише властивості, а також гетери для сетери для них. Фактично, простір імен «OpenApi» – це об’єктно-орієнтовна проекція специфікації OpenAPI.

В просторі імен «Symfony» містяться три класи. Всі вони призначені для інтеграції з Symfony. Найголовніший з них – RestDocsBundle – наслідує клас Symfony\Component\HttpKernel\Bundle\Bundle. Для того, щоб використовувати бібліотек,у потрібно додати саме цей клас-бандл в файл «config/bundles.php». Клас RestDocsBundle пустий і не містить жодної логіки.

В просторі імен «Symfony/DependencyInjection» містяться два файли – Configuration та RestDocsExtension.

В класі Configuration, який реалізує інтерфейс Sushchyk\SymfonyRestDocs\Symfony\DependencyInjection\ConfigurationInterface,відбувається формування дерева конфігурації, тобто описуються можливі параметри конфігурації. Їх можна переглянути за допомогою консольної команди «config dump:reference rest_docs». «rest_docs» – це перетворена з Camel Case в Snake Case назва бандлу «RestDocs».

В класі RestDocsExtension, який наслідує Symfony\Component\HttpKernel\DependencyInjection\Extension, в методі load() відбувається читання цієї конфігурації. Потім конфігурацією заповнюються параметри контейнеру «swagger.output» та «swagger.docs». Пізніше під час першого виклику методу send() класу Request ці параметри читаються та пишуться в Swagger-файл.

В просторі імен «Utils» міститься логіка формування об’єктів з просторі імен «OpenAPI». Там містяться такі класи як DataCompatibleWithSchemaValidator, HeadersCreator, OperationsMerger, SchemaCreatorFromArray, SchemaCreatorFromSwaggerFile. Клас DataCompatibleWithSchemaValidator відповідає за те, щоб перевіряти, чи відповідають дані, які повернулись у тілі відповіді, схемі в документації. Клас HeadersCreator займається інстанціюванням об’єктів типу Headers з масивів. Ця логіка винесена в окремий клас, оскільки зустрічається в декількох місцях. Клас OperationsMerger відповідає за формування з двох тестів, які переходять на один і той же ендпоінт, одної операції. Класи

SchemaCreatorFromSwaggerFile, SchemaCreatorFromArray відповідають за створення об’єкту типу Schema, який серіалізується в JSON та записується в Swagger-файл, відповідно з файлу та конфігурації.

3.4. Висновки

Такі мови програмування як PHP, Ruby та C# мають свої переваги та недоліки.

PHP – проста та універсальна мова програмування, яка найчастіше використовується у веб-програмуванні, але не має строгої типізації та володіє складним синтаксисом. C# та ASP.NET – стек технологій, який надає багато функцій «з коробки», однак створення застосунків з використанням цієї технології дорого коштує. Ruby – наймолодша з трьох мова програмування, яка відзначається високою ефективністю розробки програм та хорошою спільнотою, однак ця спільнота – відносно невелика – як наслідок, екосистема Ruby не дуже розвинута.

Через свою універсальність, гнучкість та простоту для методу створення документації на основі тестів була обрана мова програмування PHP.

Два найпопулярніших та найперспективніших фреймворки в екосистемі PHP – Laravel та Symfony. Laravel – високорівневий, насичений функціональністю та призначений для швидкого створення веб-застосунків. Symfony – фреймворк, який водночас є екосистемою та набором компонентів, які можна перевикористовувати, має високий поріг входу та призначений для створення складних та масштабованих застосунків. Оскільки тести частіше пишуться в проектах, розроблених за допомогою Symfony, то й для запропонованогометоду був обраний цей фреймворк.

Програмна реалізація методу у вигляді бібліотеки під назвою «Symfony Rest Docs» надає можливість задати параметри вихідної документації шляхом створення конфігураційного файлу rest_docs.yaml в

директорії config/packages/test, а також пропонує використовувати клас Request. Через yaml-конфігурацію можна задати всі параметри для вихідного Swagger-файлу, окрім «paths». Параметр «paths» збирається під час запуску інтеграційних тестів, а точніше нова операція додається в документацію під час виклику методу send() класу Request.

4. ТЕСТУВАННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ ЗАПРОПОНОВАНОГО МЕТОДУ

4.1. Тестування запропонованого методу

Тестування методу, а точніше, його програмної реалізації у вигляді бібліотеки, відбувалось з використанням автоматичних тестів, написаних за допомогою PHPUnit.

PHPUnit – фреймворк для модульного тестування під час розроблення ПЗ на PHP. Є представником сімейства фреймворків XUnit на основі пакету SUnit, створеного Кентом Беком [35]. PHPUnit розроблений Себастьяном Бергманом [36]. PHPUnit був створений з позиції – чим раніше ви виявите помилки в коді, тим швидше ви зможете їх виправити.

PHPUnit надає програмісту такі можливості:

- інструменти для створення модульних тестів і організації їх в ієрархічні набори;
- інтерфейс командного рядка для тестування;
- постачальники даних – генератори для тестування даних для перевірки, як єдиний тест поводиться на різних вхідних даних;
- підтримка тестування коду, що використовує базу даних;
- можливість тестування винятків;
- підтримка фіктивних об'єктів;
- генератор звітів;
- інтеграція з інструментом Selenium RC для тестування користувальницьких інтерфейсів.

Всього в проекті було написано два функціональні тести та один модульний. Функціональні тести повністю перевіряють працездатність програмного продукту. Зазвичай, цим типом тестів набагато простіше покривати готовий продукт, ніж модульними, так як легше зрозуміти, що саме повинна і не повинна робити певна частина призначеного для користувача інтерфейсу, ніж визначити що повинна робити дана функція. І

найбільша перевага – можна покривати функціональними тестами лише критичну функціональність – і вони будуть справно гарантувати її працездатність. Натомість модульні тести, або як їх ще називають, – юніт-тести виконують тестування окремих частин продукту, зазвичай окремих функцій або методів. Метою модульного тестування є ізоляція кожної частини програми та впевненість у тому, що кожна окрема частина є коректною [37]. Модульний тест забезпечує жорсткий «контракт», за яким має працювати тестований код. Як результат, це надає деякі переваги. Модульне тестування допомагає знайти помилки раніше в циклі розроблення ПЗ, що робить розробку дешевшою та швидшою.

Отже, перший функціональний тест перевіряє випадок, коли Swagger-файл вдало генерується на трьох різних наборах даних. Запускати один і той же тест на трьох наборах даних дозволяють провайдери даних (Data Providers) – механізм PHPUnit, який дозволяє не писати багато тестів у випадку, якщо тестується один і той же клас чи метод, натомість він запускає тест один раз на різних наборах даних. Провайдером даних є функція, яка повинна повертати двовимірний масив. Довжина цього масиву – кількість наборів даних, довжина елементів – кількість параметрів. Для того, щоб використати Data Provider, потрібно над назвою функції поставити відповідну анотацію з назвою провайдера. Для першого функціонального тесту провайдер має назву «successGenerationDataProvider». Тест приймає такі параметри як масив запитів, масив відповідей, конфігурація та очікуваний результат. Потім він проходить по всім запитам в циклі та для кожного запиту, тобто об’єкту типу Request викликає метод send(). При цьому для багатьох класів використовуються mock-об’єкти – тип об’єктів, які реалізують задані аспекти модельованого. В даному випадку в ролі модельованого оточення виступає клас KernelBrowser, який передається в метод send() як параметр. Він має змодельований метод getContainer(), який повертає такий об’єкт класу Container, що повертає конфігурацію, яка вказана в параметрі. Далі

тест зчитує вихідний файл та робить твердження, що його вміст відповідає очікуваному результату, який є в тесті, як параметр функції.

В другому функціональному тесті використовується невдалий сценарій – коли метод send() повертає не ті дані, які очікуються. Тут використовуються два набори даних – перший, коли повертається невірний код відповіді, другий – схема даних не відповідає очікуваній. Загалом, в невдалому сценарії використовуються такий же алгоритм дій, як і у вдалому за винятком того, що виконується твердження на помилку, а не на результат.

Модульним тестом покритий клас DataCompatibleWithSchemaValidator. Він запускається на 10 різних наборів даних. Результати запуску тестів зображені на рис. 4.1.

```
sushchyk@sushchyk-Vostro-15-3568:~/Projects/task-manager$ ./vendor/b
in/phpunit vendor/sushchyk/symfony-rest-docs/tests/ --coverage-html=
test
PHPUnit 8.4.0 by Sebastian Bergmann and contributors.

...............                                                   15
 / 15 (100%)

Time: 357 ms, Memory: 10.00 MB

OK (15 tests, 28 assertions)

Generating code coverage report in HTML format ... done [197 ms]
```

Рис. 4.1. Результат запуску тестів

Всього тестами покрито 65,29% рядків коду, 66.14% класів та методів та 37.04% класів та трейтів. Такі невеликі відсотки пов’язані з тим, що багато коду, якого немає сенсу покривати тестами, знаходиться в просторі імен «Symfony». Звіт по покриттю коду генерувався при запуску тестів за допомогою параметру «coverage-html» та розширення XDebug. Він зображений на рис. 4.2.

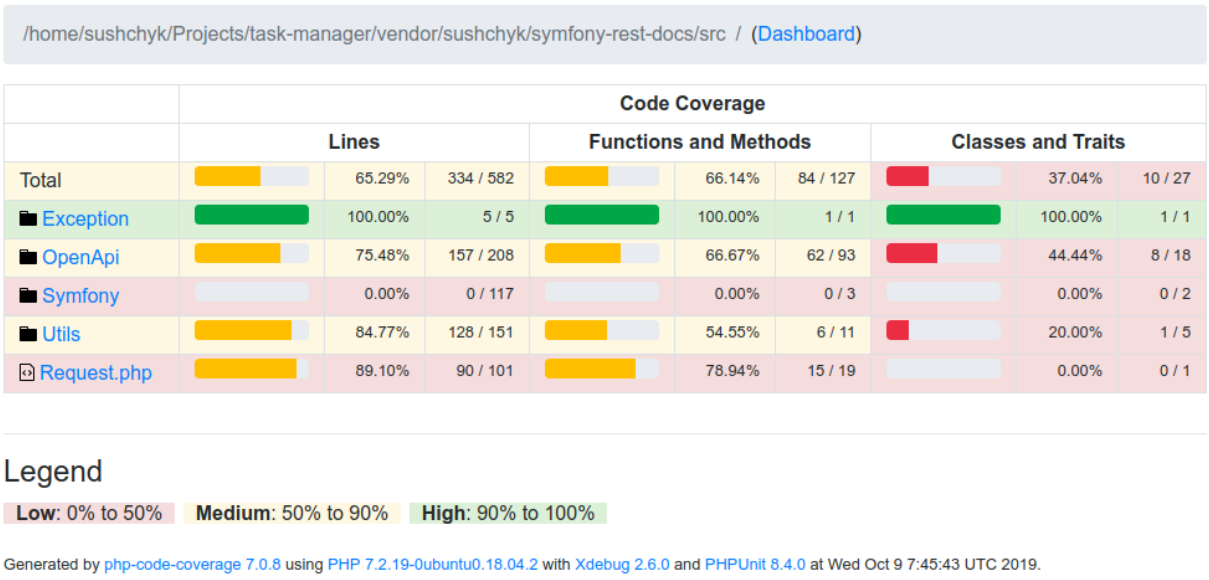


Рис. 4.2. Звіт з покриття коду тестами

4.2. Оцінка ефективності запропонованого методу

Для оцінки ефективності був написаний проект, який надає API, для нього була згенерована документація різними способами. Проект містить дві сутності – користувач та задача. Для цих сутностей доступні операції з табл. 4.1.

Таблиця 4.1

Операції в застосунку для оцінки ефективності

Метод	URL	Опис
POST	api/registration	Реєстрація користувача. Приймає як параметр логін та пароль. У разі успіху повертає код 200 та JWT-токен, згенерований для зареєстрованого користувача. У разі помилок валідації повертає 400-ий код.
POST	api/login_check	Авторизація. Як і попередній ендпоінт, приймає як параметр логін та пароль, і також повертає токен у разі успішної авторизації.. Якщо ж дані невірні, повертає 401-у помилку.

Продовження табл. 4.1

GET	/api/tasks	Повертає список задач користувача та 200-ий код. Доступні такі GET-параметри як «order_by» і «order_alghoritm».
POST	/api/tasks	Створення задачі. Приймає як параметр назву задачі та опис. Повертає 200-у помилку та об’єкт, який також містить ідентифікатор задачі у разі успіху, та помилку 400 у випадку помилки валідації.
PUT	api/tasks/{id}	Оновлення задачі. Повертає 404-у помилку у випадку, якщо задачу з таким ідентифікатором не знайдено, помилку 400, якщо провалилась валідація, та 200-ий код в разі успішного завершення.
DELETE	/api/tasks/{id}	Видалення задачі. Повертає 404-у в разі, якщо задачі з таким ідентифікатором не існує та код 200 в разі успішного видалення.

Для даного проекту було створена документація методом використання бібліотеки для конкретного фреймворку. Для цього був використаний пакет NelmioApiDocBundle для Symfony. Пакет NelmioApiDocBundle дозволяє генерувати документацію у форматі OpenAPI (Swagger) і забезпечує пісочницю для інтерактивного експериментування з API. Цей пакет підтримує маршрутизацію Symfony, анотації PHP, анотації Swagger-Php, анотації FOSRestBundle та програми, що використовують Api-Platform [38]. Для того, щоб встановити пакет, достатньо просто в консолі зайти в директорії з проектом та виконати консольну команду «composer require nelmio/api-doc-bundle». Якщо не використовується Flex, то також потрібно додати відповідний бандл в config/bundles.php, зареєструвати роут, та налаштувавати пакет.

Після того, як пакет налаштований, документація буде доступна за роутом, вказаним в конфігурації. Отже, для кожної операції над методом

контролера напишемо відповідні анотації, які містять опис операції. Наприклад, для створення задачі це виглядатиме так як на лістингу 4.1.

Лістинг 4.1. Приклад анотацій для Swagger-документації

```
  * @SWG\Parameter(
     *     name=«title",
     *     in="body",
     *     type="string",
     *     description="Task title"
     * )
     * @SWG\Parameter(
     *     name="description",
     *     in="body",
     *     type="string",
     *     description="Task description"
     * )
     * @SWG\Response(
     *     response=200,
     *     description="Returns task",
     *     @SWG\Schema(type="object",
     *                                     @SWG\Property(property="task",
ref=@Model(type=Task::class))
     *     )
     * )
```

Також для того, щоб можна було об’єктивно порівняти цей метод з методом використання інструментів для конкретного фреймворку чи мови програмування потрібно написати інтеграційні тести, адже як було сказано у розділі 1, саме тоді затрати часу будуть приблизно однаковими. Тести будуть просто перевіряти код відповіді, як це показано на лістингу 4.2.

Лістинг 4.2. Приклад тестів для API

```
/**
 * @test
 */
public function itStoresTask()
{
    $taskTitle = 'Task Title';
    $taskDescription = 'Task Description';

    $response = $this->jsonRequest('POST', '/api/tasks', [], [
        'title' => 'Title',
        'description' => "Description"
    ], $this->getUser());

    $this->assertEquals(200, $response->getStatusCode());
}

/**
 * @test
 */
public                                                    function
itDoesNotStoreTaskAndReturnsValidationErrorWhenDataIsInvalid()
{
```

```
        $response = $this->jsonRequest = $this->jsonRequest('POST',
'/api/tasks', [], [
            'title' => '',
            'description' => ''
        ], $this->getUser());

        $this->assertEquals(400, $response->getStatusCode());
    }
```

Час – доволі суб'єктивний критерій, адже в когось може бути використано менше часу на додавання анотацій та написання тестів, а в когось більше. Тому будемо вважати, що програмісти в середньому пишуть n рядків за годину, і порівняємо кінцеву кількість рядків коду в обох методах. Отже, загалом тести та анотації при використанні NelmioApiDocBundle займають 545 рядків коду в даному проекті. Документація виглядає так, як показано на рис. 4.3.

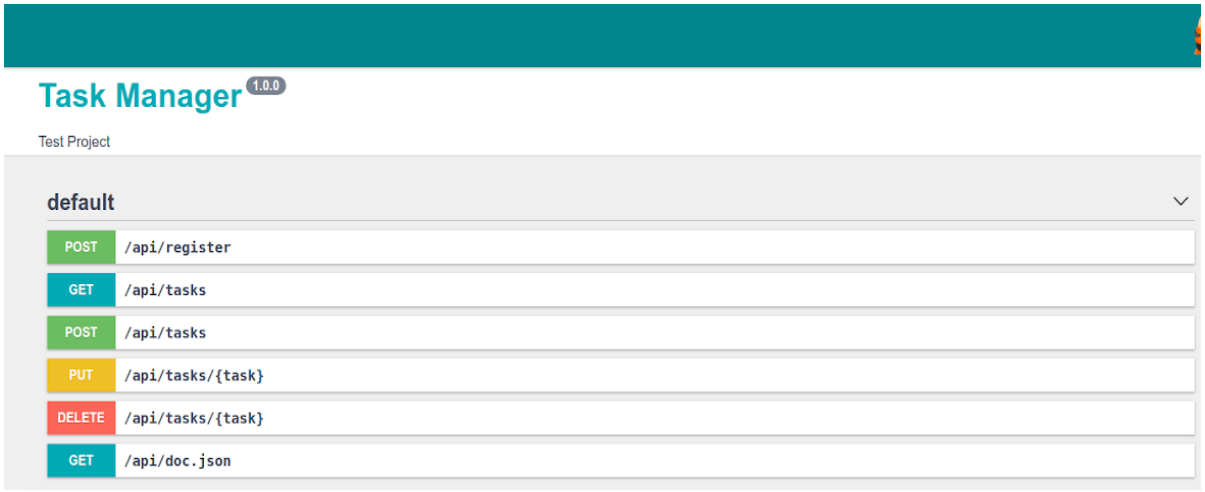


Рис. 4.3. Swagger-документація для проекту

Натомість при використанні запропонованого методу тести будуть виглядати як на лістингу 4.3, та всього будуть займати 411 рядків. Також документація буде такою ж як і на рис. 4.3 та інтерактивною, чого не забезпечує існуюча реалізація методу на основі тестів Spring Rest Docs.

Лістинг 4.3. Приклад тесту при використанні Symfony Rest Docs

```
public function itStoresTask()
{
    $taskTitle = 'Task Title';
    $taskDescription = 'Task Description';

    $request = new Request('POST', '/api/tasks');
```

```
        $request
            ->withHeader(
                'Authorization',
                ['description' => 'Auth token. Example: `Bearer
{token}`'],
                $this->getAuthorizationHeaderValue($this->getUser())
            )
            ->withBody(
                [
                    'title'       => $taskTitle,
                    'description' => $taskDescription
                ],
                [
                    'type'       => 'object',
                    'required'   => [
                        'name',
                        'description'
                    ],
                    'properties' => [
                        'name'        => [
                            'type' => 'string'
                        ],
                        'description' => [
                            'type' => 'string'
                        ]
                    ]
                ]
            )
            ->withTags(['Tasks'])
            ->withDescription('Store task')
            ->expectsResponse(200, [
                'description' => 'Task has been successfully stored'
            ]);

        $request->send(static::createClient());
    }
```

4.3. Висновки

Отже, код бібліотеки покритий інтеграційними та модульними тестами написаних на PHPUnit з використанням провайдерам даних. Всього тестів – 15, а тверджень в них – 28. Тестами покрито близько 65% рядків коду та 66% функцій та методів.

Для оцінки ефективності методу було написано проект з шістьма операціями. Для цього проекту ми спочатку створили документацію за допомогою використання методу створення документації інструментами для конкретного фреймворку, де у ролі інструменту виступав пакет NelmioDocApiBundle, а у ролі фреймворку – Symfony. Потім була створена документації запропонованим методом, і в результаті виявилось, що якщо враховувати тести, то метод на основі тестів потребує менше часових

затрат, оскільки таким способом документація потребує написання меншої кількості рядків коду. Також запропонований метод забезпечує інтерактивність документації – тобто надає «пісочницю», де користувачі API можуть експериментувати з ним, що не реалізовано в Spring Rest Docs.

5. ПОБУДОВА БІЗНЕС-МОДЕЛІ

5.1. Аналіз та опис проблеми

В наш час великими темпами зростають вимоги до якості та швидкості розроблення програмного забезпечення. Продукт, який динамічно розвивається та володіє великим об’ємом функціональності, неможливо підтримувати без таких артефактів як тести та документація. Обидва артефакти дають можливість технічним спеціалістам зекономити час на розумінні того, як працює продукт, в розробленні якого вони беруть участь. Водночас, створення та підтримка якісних тестів та документації потребують часових затрат програмістів та аналітиків, які для бізнесу перетворюються на фінансові.

У веб-програмуванні все більше популярності набирає відокремлення клієнтської частини від серверної за допомогою технології односторінкових додатків. Натомість, серверну частину розбивають на мікросервіси, що дозволяє зменшити залежність між функціональними компонентами сайту та розподілити їх розроблення між людьми або групами людей. Взаємодія клієнтської та серверної частини, а також мікросервісів в такому випадку відбувається за допомогою API. Частіше всього над різними компонентами системи працюють окремі команди розробників. Інколи ці команди фізично знаходяться не в одному офісі, і навіть не в одному місті. Саме тому, будь-яке API потребує якісної документації.

На даний момент для створення документації для REST API частіше всього використовуються чотири методи: створення документації за допомогою інструментів для конкретної мови програмування або фреймворку, створення документації вручну, створення документації за допомогою сторонніх утиліт та створення документації на основі тестів. Кожен з них має ряд суттєвих недоліків.

У розробці API доволі часто трапляються випадки, коли робота API не відповідає документації, тому що якийсь розробник забув внести зміни в документацію. Існують механізми, які вирішують такі проблеми, і не дають можливості опинитись в такій ситуації, але не всі методи мають змогу впровадити такі механізми. Саме тому, на мою думку, метод створення документації повинен максимально захищати розробників від даної проблеми шляхом автоматичного чи напівавтоматичного внесення змін в документацію після зміни логіки роботи програми. Тобто, «людський фактор» повинен бути мінімізований. Також головна проблема будь-якої документації – це час на її розроблення, що повинна бути враховано в методі, адже це чи не найважливіший критерій, за яким обирають той чи інший метод. Також важливо, щоб документація була інтерактивною – тобто, користувачі API прямо в своєму браузері могли авторизовуватись та відправляти запити до серверу, адже це дає можливість не встановлювати стороннє програмне забезпечення для таких цілей.

Аналіз наявних методів показав, що жоден з них повністю не розв’язує вище описаних проблем. Метод створення документації за допомогою інструментів для конкретної мови програмування або фреймворку потребує невеликих часових витрат, але не завжди дозволяє створювати інтерактивний графічний інтерфейс та не забезпечує актуальності. Використання сторонніх утиліт також дозволяє суттєво зекономити час, і навіть надає графічний інтерфейс, але не вирішує проблему з «людським фактором» та має ряд інших проблем, адже за повноцінне використання сторонніх утиліт потрібно платити [39]. Метод написання документації вручну взагалі не вирішує жодної з проблем. Натомість, метод на основі тестів єдиний з всіх дозволяє усунути «людський фактор». Якщо на проекті використовуються функціональні тести, то він, фактично, не потребує додаткових часових витрат.

Натомість, його наявна реалізація не здатна генерувати інтерактивну документації.

Всі проблеми наявних методів узагальнені на дереві проблем, яке зображене на рис. 5.1.

Рис. 5.1. Дерево проблем

5.2. Зацікавлені сторони

У подоланні проблем, які описані вище, існує декілька зацікавлених сторін.

На мою думку, найбільше всього у вирішенні даних проблем зацікавлені безпосередньо розробники програмного забезпечення, адже саме вони щоденно стикаються з ними. На їх вирішення вони витрачають велику кількість свого робочого часу, який вони могли використати більш раціонально. Багато з них віддають перевагу іншим стандартам для API – таких, як GraphQL або JsonRPC, тому що їх використання вирішує ці проблеми, натомість, породжує ряд інших, які для розробників є не такими

неприємними. Натомість, вплив рядових розробників не суттєвий, адже все, що вони можуть – це поскаржитись на проблеми головному технічному спеціалісту в команді, бо лише він може ініціювати зміну вже існуючих процесів в розробці продукту.

Наступною зацікавленою стороною є середня ланка менеджменту, до якої відносяться керівники команди та проектні менеджери. Вони в щоденній роботі також стискаються з проблемами, описаними в підрозділі 5.1, та їх наслідками. Зокрема, через неякісну документацію часто виникають проблеми з взаємодіями команд, компоненти системи яких взаємодіють через API. Через неправильну документацію на одній із сторін може бути невірно реалізована певна функціональність, що загрожує додатковими часовими витратами на розробку, і, відповідно, зриву певних планів та графіків. Вплив даної групи зацікавлених осіб є досить великим, оскільки саме вони відповідальні за налагодження процесів, що включає в себе і вибір методу для створення та підтримки документації для API.

Ще однією зацікавленою особою є власники та замовники програмного забезпечення. Зазвичай, вони дуже рідко розуміють безпосередні технічні деталі та процеси. Більше того, даний тип документації їх не цікавить – оскільки вона потрібна лише технічним спеціалістам для розроблення продукту, а не кінцевим користувачам. Але, водночас, всі зазначені проблеми, так чи інакше призводять до фінансових витрат та часових затримок при розробленні продукту, а у вирішенні таких проблем дана категорія осіб зацікавлена набагато більше за інших. Вплив даної зацікавленої особи у вирішенні проблем, наведених в підрозділі 5.1 не суттєвий, адже вони, зазвичай, не беруть участь у щоденних технічних процесах – цим займається середня ланка менеджменту.

У табл. 5.1 наведено всі групи зацікавлених сторін, їх інтереси та вплив (міра зацікавленості у вирішенні наявних проблем).

Таблиця 5.1

Зацікавлені сторонни

Зацікавлена сторона	Інтерес зацікавленої особи	Вплив зацікавленої особи	Стратегії приваблення зацікавлених сторін
Розробники програмного забезпечення	Мінімізація часових затрат на створення та підтримку документації для API та її інтерактивність	низький	Проведення презентацій, публікація матеріалів на спеціалізованих ресурсах, виступи на конференціях
Середня ланка менеджменту	Забезпечення актуальності документації, що дозволяє уникнути проблемних ситуацій	високий	
Власники та замовники програмного забезпечення	Мінімізація фінансових витрат, висока швидкість впровадження нової функціональності	середній	

5.3. Комерційне рішення. Основні характеристики

Відповідно до вищезазначених проблем, можна описати кінцевий продукт, що буде їх вирішувати. Це буде новий метод для створення документації для REST API, а точніше вдосконалена версія методу на основі тестів. Він буде реалізований за допомогою з найпопулярніших мов програмування, але, фактично, даний метод можна буде легко перенести на будь-яку іншу мову. Він не буде прив'язаний до якогось фреймворку. Це буде продукт у вигляді розширення для програми, тобто, бібліотеки, яка буде вже інтегрована з найпоширенішою бібліотекою для юніт-тестування. Наприклад, у випадку з Java – це буде JUnit, PHP – PHPUnit, Python – pytest.

Даний продукт повинен мати обширну документацію, яка містиме список програмних методів, конфігурації та функціональних можливостей.

Також там буде інформація про те, як долати вище описані проблеми за допомогою інтеграції з системами неперевного розгортання, Swagger-клієнтів тощо.

Цей продукт повинен бути простим у використанні, але водночас мати багато функціональних можливостей, серед яких варто виділити – можливість авторизуватись, яка буде реалізована за допомогою парсингу конфігурації при генеруванні Swagger-файлу, можливість додавання коментарів та опису до запитів, які надає сервер, щоб документація для API була детальною.

Очевидно, що клієнтом даного продукту є компанії-розробники програмного забезпечення, які працюють над його серверною частиною, яка використовує REST API для надання доступу до інформаційних даних. Тобто монетизація даного продукту буде побудована на моделі співробітництва бізнесу для бізнесу (B2B).

5.4. Конкурентні переваги рішення

Наявні методи створення документації мають суттєві недоліки та не вирішують широкий ряд проблем. Найголовнішою, на мою думку, є присутність у процесі можливості розходження роботи серверної частини та документації до API. Цю проблему може вирішити лише метод на основі тестів. Наразі існує лише одна реалізація даного методу – Spring Rest Docs, яка є бібліотекою для мови програмування Java. Але її недолік полягає в тому, що вона генерує не інтерактивну графічну документацію, а використовує власний формат для відображення документації. Даний недолік буде усунено, і результатом прогону тестів буде файл, який відповідає відомому формату OpenAPI.

Отже, підсумовуючи, конкурентними перевагами даного продукту є:

- мінімальні затрати на створення та підтримку документації, якщо на проекті використовуються інтеграційні тести;
- підвищення технічної якості вихідного продукту шляхом

написання інтеграційних тестів;

- забезпечення актуальності документації та автоматичного внесення змін в документацію за допомогою обов'язкового написання тестів на нову функціональність;
- забезпечення інтерактивності документації за допомогою генерації Swagger-файлу, який відповідає стандарту OpenAPI.

5.5. Клієнти. Сегменти ринку споживання

Як зазначено вище, клієнтами даного продукту є компанії-розробники програмного забезпечення.

Насамперед, можна провести сегментацію компаній за типом. За цією ознакою їх можна поділити на два типи – продуктові та аутсорсингові компанії [40]. До першої категорії відносять компанії, які є власниками продуктів, які їх розробляють. Натомість, аутсорсингові розробляють продукти на замовлення або перепродують програмні продукти своїх найманих працівників іншим роботодавцям. Я вважаю, що друга категорія компаній навряд чи буде зацікавлена у даному продукті. Часто команди розробників в аутсорсингових компаніях працюють над розробленням мінімально життєздатної версії продуктів, і якість технічної документації для цих продуктів не грає абсолютно ніякої ролі. Натомість, продуктові компанії більше зацікавлені в покращенні процесів та якості програмного забезпечення.

Наступна ознака – це об'єм компанії. В невеликих компаніях добре налагоджена комунікація між різними командами, тому будь-яка погана технічна документація перекривається можливістю зачасту особисто задати питання розробнику щодо того, як потрібно використовувати API. Натомість великі компанії зазвичай працюють над великими за обсягом функціональності та коду проектах, намагаються розділити відповідальність між командами за різні компоненти системи та частіше стикаються з проблемами, які вирішує даний продукт.

Ну і остання доволі суб’єктивна ознака – це технологічна зрілість компанії. Є навіть доволі успішні та великі проекти, при розробленні яких не використовують тестів, складної архітектури та API. Компанії, проекти яких перебувають на такій стадії, теж навряд чи можна буде зацікавити розробленим методом для створення документації. Та варто зазначити, що це тимчасово, адже рано чи пізно вони зіткнуться з необхідністю вертикального масштабування, а на такому етапі тести та документація є дуже важливими артефактами.

Отже, в розробленому продукті, найбільше всього будуть зацікавлені великі за об’ємом технологічно зрілі продуктові компанії.

5.6. Унікальна ціннісна пропозиція

Унікальна ціннісна пропозиція – це пояснення того, як продукт вирішує проблему. Унікальну ціннісну пропозицію можна представити у вигляді суми проблеми та її рішення.

При формуванні дерева проблем було виявлено головні проблеми, з якими доводиться стикатися при створенні документації для REST API, а при аналізі зацікавлених сторін – очікування відповідних сторін від продукту, який буде їх рішенням. Безпосередньо рядові розробники хочуть мінімізувати час, який витрачається на створення та підтримку документації для API, та хочуть щоб документація була інтерактивною. Для середньої ланки менеджменту головне – забезпечення актуальності документації, що дозволяє уникнути проблемних ситуацій. Власників та замовників програмного забезпечення цікавить мінімізація фінансових витрат, висока швидкість впровадження нової функціональності. Запропоноване рішення дозволяє задовольнити всі вимоги зацікавлених сторін.

Отже, унікальною ціннісною пропозицією є бібліотека для створення документації для REST API, що генерує інтерактивну документацію під час запуску інтеграційних тестів.

5.7. Доходи та витрати

Сумарний від даного продукту буде складатись з доходів від продажу ліцензій на його використання в проектах та доходів від консультацій та підтримки клієнтів.

Реалізований продукт буде продаватись як закрите програмне забезпечення компаніям-розробникам програмного забезпечення. З клієнтами буде підписуватись ліцензійна угода, згідно якої у них будуть обмежені права на використання бібліотеки – зокрема, буде заборонено її модифікація, розповсюдження та перепродаж. Після цього клієнти отримуватимуть доступ до сирцевого коду та документації. Також, при бажанні, вони зможуть придбати консультації розробників, які допоможуть їм як найефективніше використовувати бібліотеку в своїх цілях та підтримку ПЗ, що означатиме можливість оновлення до нових версій, які міститимуть ще більше функціональних можливостей.

Витрати ж на розробку даного продукту незначні. На перших етапах для реалізації продукту в життя буде достатньо одного або двох кваліфікованих програмістів. Також перед початком продажу продуктів знадобляться юридичні та бухгалтерські послуги та офіс.

З витратами на реалізацію проекту та прогнозованими прибутками можна ознайомитись в табл. 5.2 та 5.3.

Таблиця 5.2

Витрати та прогнозовані проекти на перше півріччя ($)

Найменування	Місяць	Місяць	Місяць	Місяць	Місяць	Місяць

витрат	1	2	3	4	5	6
Загальні витрати	2100	4200	5100	6000	7100	7200
З/П розробників	2000	3500	3500	4000	5000	5000
Юр. та бухг. послуги	–	500	1000	1000	1100	1200
Реклама	100	200	300	300	400	400
Господарські витрати	–	200	300	500	500	500
Заплановані прибутки	–	–	1300	2300	4400	5400
Результат (без врахування податків)	−2100	−4200	−3800	−2500	−2600	−1700

Таблиця 5.3

Витрати та прогнозовані проекти на друге півріччя ($)

Найменування витрат	Місяць 7	Місяць 8	Місяць 9	Місяць 10	Місяць 11	Місяць 12
Загальні витрати	8200	8200	8500	9500	9500	9500
З/П розробників	6000	6000	6000	7000	7000	7000
Юр. та бухг. послуги	1200	1200	1400	1400	1400	1400
Реклама	500	500	500	500	500	500
Господарські витрати	500	500	600	600	600	600
Заплановані прибутки	6000	8000	10500	12500	14500	15500

Продовження табл. 5.2

Результат	−2200	−200	+2000	+3000	+5000	+6000

5.8. Результат побудови бізнес-моделі

Узагальнимо все написане вище у лаконічну бізнес-модель у вигляді lean canvas.

Споживачі: великі технологічно зрілі продуктові компанії-розробники програмного забезпечення.

Проблема: великі часові затрати на створення документації для REST API, її невідповідність роботі серверної частини, поганий та не інтерактивний графічний інтерфейс для документації.

Рішення: оптимізований метод на основі тестів, який забезпечує інтерактивний графічний інтерфейс.

Унікальна ціннісна пропозиція: бібліотека для створення документації для REST API, що генерує інтерактивну документацію під час запуску інтеграційних тестів.

Потоки доходів: продаж ліцензій на використання програмного забезпечення клієнтами в їх проектах та консультація та підтримка програмного забезпечення для клієнтів.

Структура витрат: заробітна плата розробникам продукту, юридичні та бухгалтерські послуги, господарські витрати (оренда та підтримка у належному стані приміщення), реклама, податки.

Також в канву бізнес-моделі включаються структурні блоки: прихована перевага (перевага, яку неможливо скопіювати або купити), ключові метрики (основні показники, що вимірюються) та канали (шляхи до користувачів).

Канали: контекстна реклама та реклама у вигляді публікацій на спеціалізованих ресурсах.

Ключові метрики: кількість проданих ліцензій.

Прихована перевага: простота у користуванні, обширна та якісна документація.

Бізнес-модуль наведена у зведеному вигляді у таблиці 5.4.

Отже, зважаючи на дані у таблиці 5.4, можна зробити висновок, що запропонований проект, який реалізує описаний у дисертації метод створення документації для REST API має певні перспективи успішної реалізації. Варто зазначити, що дана модель не враховує багатьох факторів та ризиків.

Таблиця 5.4

Канва бізнес-моделі

<table>
<tr><td rowspan="2">Проблема

великі часові затрати на створення документації для REST API

її невідповідність роботі серверної частини

неінтерактивний графічний інтерфейс для документації</td><td>Рішення

оптимізований метод на основі тестів, який забезпечує інтерактивний графічний інтерфейс</td><td rowspan="2">Унікальна ціннісна пропозиція

бібліотека для створення документації для REST API, що генерує інтерактивну документацію під час запуску інтеграційних тестів</td><td>Прихована перевага

простота у користуванні, обширна та якісна документація</td><td rowspan="2">Споживачі

великі технологічно зрілі продуктові компанії, які є розробниками програмного забезпечення</td></tr>
<tr><td>Ключові метрики

кількість проданих ліцензій</td><td>Канали

контекстна реклама та реклама у вигляді публікацій на ресурсах</td></tr>
</table>

Продовження табл. 5.4

<table>
<tr><td>Структура витрат</td><td>Потоки доходів</td></tr>
</table>

заробітна плата розробникам продукту, юридичні та бухгалтерські послуги, господарські витрати (оренда та підтримка у належному стані приміщення), реклама, податки.	продаж ліцензій на використання програмного запезпечення клієнтами в їх проектах та консультація та підтримка програмного забезпечення для клієнтів

5.9. Висновки

У даному розділі було проведено аналіз створення та підтримки документації для REST API, було виявлено основні проблеми цього процесу, які не вирішують наявні методи та проаналізовані вимоги, інтереси та вплив зацікавлених сторін. Внаслідок цього було запропоновано комерційне рішення, яке вирішує виявлені проблеми, задовольняє вимоги зацікавлених сторін та має ряд конкретних переваг. Був здійснений аналіз майбутніх клієнтів та дослідження сегментів ринку споживання. Це дозволило спрогнозувати потенційні доходи та витрати на реалізацію продукту. У результаті була створена бізнес-модель, що обґрунтовує доцільність реалізації даного продукту та прогнозує його прибутковість в майбутньому.

ВИСНОВКИ

У даній магістерській роботі виконані наступні завдання:

1. Проведено аналіз таких стандартів для створення API як REST, GraphQL, SOAP, JSON RPC та обґрунтовано що проблема процесу створення документації актуальна саме для REST.

2. Сформовано вимоги до документації для REST API – актуальність, інтерактивність та мінімальні часові затрати на її створення та підтримку.

3. Проаналізовано наявні методи для створення документації для REST API – створення документації за допомогою інструментів для конкретної мови програмування або фреймворку, створення документації вручну, створення документації за допомогою сторонніх утиліт, створення документації на основі тестів. Найбільшій кількості вимог задовольняє метод створення документації на основі тестів.

4. Проаналізовано наявну реалізацію методу на основі тестів – Spring Rest Docs та виявлено її основні недоліки. До основних недоліків належать неінтерактивність документації та необхідність внесення змін в шаблони при додаванні нових тестів.

5. Запропоновано власний метод на основі тестів, який усуває недоліки наявної реалізації за допомогою використання стандарту OpenAPI.

6. Розроблено програмну реалізацію методу з використанням таких технологій як PHP та Symfony.

7. Програмна реалізація була покрита автоматичними тестами, написаними з використанням бібліотеки PHPUnit. Загальний відсоток покриття коду тестами сягає 65%.

8. Проаналізовано ефективність методу в порівнянні з іншими методами та наявною реалізацією методу на основі тестів. Запропонований метод забезпечує інтерактивність документації в порівнянні з Spring Rest

Docs та потребує на 25% відсотків менше часу ніж метод створення документації інструментами для конкретного фреймворку.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Learn GraphQL [Електронний ресурс] – Режим доступу: https://graphql.org/learn// – Дата доступу: 10.02.2019 – Назва з екрану.
2. GraphQL Core Concepts Tutorial [Електронний ресурс] – Режим доступу: https://www.howtographql.com/basics/2-core-concepts/ – Дата доступу: 10.02.2019 – Назва з екрану.
3. Преимущества GraphQL | Blog Meline [Електронний ресурс] – Режим доступу: https://meline.lviv.ua/development/other/graphql/ – Дата доступу: 10.02.2019 – Назва з екрану.
4. Wieruch, R. The Road to Graphql: Your Journey to Master Pragmatic Graphql in JavaScript with React.Js and Node.Js [Text] / Wieruch Robbin – 1st edition. – New York: Independently published, 2018. – 314 p.
5. JSON-RPC 2.0 Specification [Електронний ресурс] – Режим доступу: https://www.jsonrpc.org/specification – Дата доступу: 10.02.2019 – Назва з екрану.
6. Implementations – JSON-RPC [Електронний ресурс] – Режим доступу: https://www.jsonrpc.org/archive_json-rpc.org/implementations.html – Дата доступу: 10.02.2019 – Назва з екрану.
7. SOAP Version 1.2 Part 0: Primer (Second Edition) [Електронний ресурс] – Режим доступу: https://www.w3.org/TR/2007/REC-soap12-part0-20070427/ – Дата доступу: 10.02.2019 – Назва з екрану.
8. WSDL Web Services Description Language [Електронний ресурс] – Режим доступу: https://www.guru99.com/wsdl-web-services-description-language.html/ – Дата доступу: 10.02.2019 – Назва з екрану.
9. Architectural Styles and the Design of Network-based Software Architectures [Електронний ресурс] – Режим доступу: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm – Дата доступу: 10.02.2019 – Назва з екрану.

10. History of REST APIs - Mobapi [Електронний ресурс] – Режим доступу: https://www.mobapi.com/history-of-rest-apis/ – Дата доступу: 10.02.2019 – Назва з екрану.
11. REST Architectural Constraints [Електронний ресурс] – Режим доступу: https://restfulapi.net/rest-architectural-constraints/ – Дата доступу: 10.02.2019 – Назва з екрану.
12. Top specification formats for REST APIs [Електронний ресурс] – Режим доступу: https://nordicapis.com/top-specification-formats-for-rest-apis – Дата доступу: 10.02.2019 – Назва з екрану.
13. Openapis. About [Електронний ресурс] – Режим доступу: https://www.openapis.org/about – Дата доступу: 10.02.2019 – Назва з екрану.
14. Swagget Specification. Basic structure [Електронний ресурс] – Режим доступу: https://swagger.io/docs/specification/basic-structure – Дата доступу: 10.02.2019 – Назва з екрану.
15. https://www.visual-paradigm.com/guide/development/swagger-vs-api-blueprint/ – Дата доступу: 10.02.2019 – Назва з екрану.
16. Is RAML or Swagger better for building APIs [Електронний ресурс] – Режим доступу: https://blog.vsoftconsulting.com/blog/is-raml-or-swagger-better-for-building-apis – Дата доступу: 10.02.2019 – Назва з екрану.
17. Symfony. NelmioApiDocBundle Docs [Електронний ресурс] – Режим доступу: https://symfony.com/doc/current/bundles/NelmioApiDocBundle/index.html – Дата доступу: 10.02.2019 – Назва з екрану.
18. What is Postman and why use it [Електронний ресурс] – Режим доступу: https://www.digitalcrafts.com/blog/student-blog-what-postman-and-why-use-it – Дата доступу: 10.02.2019 – Назва з екрану.

19. About Postman [Електронний ресурс] – Режим доступу: https://www.getpostman.com/about-postman – Дата доступу: 10.02.2019 – Назва з екрану.
20. Spring REST Docs [Електронний ресурс] – Режим доступу: https://docs.spring.io/spring-restdocs/docs/2.0.5.BUILD-SNAPSHOT/reference/html5/#getting-started – Дата доступу: 22.06.2019 – Назва з екрану.
21. What is Swagger [Електронний ресурс] – Режим доступу: https://swagger.io/docs/specification/2-0/what-is-swagger/ – Дата доступу 22.06.2019 – Назва з екрану.
22. Usage of server-side programming languages for websites [Електронний ресурс] – Режим доступу: https://w3techs.com/technologies/overview/programming_language – Дата доступу: 22.06.2019 – Назва з екрану.
23. Introduction to the C# Language and the .NET Framework [Електронний ресурс] – Режим доступу: https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework – Дата доступу: : 22.06.2019 – Назва з екрану.
24. Why PHP Development is becoming so popular [Електронний ресурс] – Режим доступу: https://www.goodworklabs.com/reasons-why-php-development-is-becoming-so-popular/ – Дата доступу: 12.10.2019 – Назва з екрану.
25. ASP.NET | Open-source web framework for .NET [Електронний ресурс] – Режим доступу: https://dotnet.microsoft.com/apps/aspnet – Дата доступу: 12.10.2019 – Назва з екрану.
26. About Ruby [Електронний ресурс] – Режим доступу: https://www.ruby-lang.org/en/about/ – Дата доступу: 12.10.2019 – Назва з екрану.
27. Laravel - The PHP Framework For Web Artisans [Електронний ресурс] – Режим доступу: https://laravel.com/docs/6.x – Дата доступу: 12.10.2019 – Назва з екрану.

28. PHP фреймворк Symfony. Обзор, плюсы, минусы, отзывы [Електронний ресурс] – Режим доступу: http://unetway.com/blog/symfony-framework-review/ – Дата доступу: 12.10.2019 – Назва з екрану.
29. OpenAPI-Specification/2.0.md at master [Електронний ресурс] – Режим доступу: https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md – Дата доступу: 12.10.2019 – Назва з екрану.
30. Introduction - Composer [Електронний ресурс] – Режим доступу: https://getcomposer.org/doc/00-intro.md – Дата доступу: 12.10.2019 – Назва з екрану.
31. PHPUnit - The PHP Testing Framework [Електронний ресурс] – Режим доступу: https://phpunit.de/ – Дата доступу: 02.11.2019 – Назва з екрану.
32. PHP Tutorial Getting Started With Composer [Електронний ресурс] – Режим доступу: https://www.codementor.io/jadjoubran/php-tutorial-getting-started-with-composer-8sbn6fb6t – Дата доступу: 02.11.2019– Назва з екрану.
33. Get Composer [Електронний ресурс] – Режим доступу: https://getcomposer.org/ – Дата доступу: 02.11.2019 – Назва з екрану.
34. Introduction - Composer [Електронний ресурс] – Режим доступу: https://getcomposer.org/doc/00-intro.md – Дата доступу: 02.11.2019 – Назва з екрану.
35. Module Testing [Електронний ресурс] – Режим доступу: https://sce.uhcl.edu/whiteta/sdp/moduleTesting.html – Дата доступу: 02.11.2019 – Назва з екрану.
36. Sebastian Bergman [Електронний ресурс] – Режим доступу: https://sebastian-bergmann.de/ pricing – Дата доступу: 02.11.2019 – Назва з екрану.

37. Jorgensen, P. Software Testing: A Craftsman's Approach [Text] / Paul C. Jorgensen – 3rd edition. – New York: Crc Press, 2018. – p. 379
38. NelmioApiDocBundle/README.md at master · nelmio/NelmioApiDocBundle · GitHub [Електронний ресурс] – Режим доступу: https://github.com/nelmio/NelmioApiDocBundle/blob/master/README.md – Дата доступу: 02.11.2019 – Назва з екрану.
39. Postman | Plans & Pricing [Електронний ресурс] – Режим доступу: https://www.getpostman.com/pricing – Дата доступу: 05.03.2019 – Назва з екрану.
40. Types of IT companies or Software Companies [Електронний ресурс] – Режим доступу: https://www.campusplusplus.com/types-of-it-companies/ – Дата доступу: 05.03.2019 – Назва з екрану.

ДОДАТКИ

Додаток 1

Приклади тексту програми

```php
<?php

namespace Sushchyk\SymfonyRestDocs\Exception;

use Throwable;

class DataConflictsWithSchemaException extends \Exception
{
    public function __construct($message, $dataName,
$defaultValueForDataName = true)
    {
        if (empty($dataName) && $defaultValueForDataName) {
            $dataName = 'data';
        }
        $message = sprintf($message, $dataName);
        parent::__construct($message);
    }
}
```

```php
<?php

namespace Sushchyk\SymfonyRestDocs\Symfony\DependencyInjection;

use Symfony\Component\Config\Definition\Builder\ArrayNodeDefinition;
use Symfony\Component\Config\Definition\Builder\TreeBuilder;
use Symfony\Component\Config\Definition\ConfigurationInterface;

class Configuration implements ConfigurationInterface
{
    /**
     * Generates the configuration tree builder.
     *
     * @return \Symfony\Component\Config\Definition\Builder\TreeBuilder The
tree builder
     */
    public function getConfigTreeBuilder()
    {
        $treeBuilder = (new TreeBuilder('rest_docs'));

        $rootNode = $treeBuilder->getRootNode();
        $rootNode
            ->children()
                ->scalarNode('output')->defaultValue('swagger.json')-
>info('Path to output file')->end();


        $swagger = $rootNode->children()->arrayNode('swagger')-
>info('Swagger configuration');

        $this->addInfoSection($swagger);

        $rootNode->end();

        return $treeBuilder;
    }

    private function addInfoSection(ArrayNodeDefinition
$arrayNodeDefinition)
    {
        $arrayNodeDefinition
            ->children()
            ->arrayNode('info')
            ->info('Provides metadata about the API. The metadata can be
used by the clients if needed.')
```

```
            ->isRequired()
                ->children()
                    ->scalarNode('title')
                        ->isRequired()
                        ->defaultValue('App')
                        ->info('The title of the application.')
                        ->end()
                    ->scalarNode('description')
                        ->info('A short description of the application.')
                        ->end()
                    ->scalarNode('termsOfService')
                        ->info('The Terms of Service for the API.')
                        ->end()
                    ->arrayNode('contact')
                        ->canBeEnabled()
                        ->info('Contact Object')
                        ->children()
                            ->scalarNode('name')
                                ->info('The identifying name of the contact
person/organization.')
                                ->end()
                            ->scalarNode('url')
                                ->info('The URL pointing to the contact
information')
                                ->end()
                            ->scalarNode('email')
                                ->info('The email address of the contact
person/organization.')
                                ->end()
                        ->end()
                        ->end()
                    ->arrayNode('license')
                        ->canBeEnabled()
                        ->info('License object')
                        ->children()
                            ->scalarNode('name')
                                ->info('The license name used for the
API.')
                                ->end()
                            ->scalarNode('url')
                                ->info('A URL to the license used for the
API.')
                                ->end()
                            ->end()
                         ->end()
                        ->scalarNode('version')
                        ->isRequired()
                        ->info('Provides the version of the application API
(not to be confused with the specification version).')
                    ->end()
                ->end()
                ->end()
                ->scalarNode('host')
                    ->info('Host (name or ip) serving the API')
                    ->end()
                ->arrayNode('schemes')
                    ->info('The transfer protocol of the API')
                    ->prototype('scalar')
                    ->end()
                    ->end()
                ->arrayNode('consumes')
                    ->info('A list of MIME types the APIs can consume.')
                    ->prototype('scalar')
                    ->end()
```

```
                    ->end()
                ->arrayNode('produces')
                    ->info('A list of MIME types the APIs can produce.')
                    ->prototype('scalar')
                    ->end()
                    ->end()
                ->arrayNode('definitions')
                    ->info('An object to hold data types produced and
consumed by operations.')
                    ->ignoreExtraKeys(false)
                    ->children()
                    ->end()
                    ->end()
                ->arrayNode('parameters')
                    ->info('An object to hold parameters that can be used
across operations.')
                    ->ignoreExtraKeys(false)
                    ->children()
                    ->end()
                    ->end()
                ->arrayNode('responses')
                    ->info('An object to hold responses that can be used
across operations.')
                    ->ignoreExtraKeys(false)
                    ->children()
                    ->end()
                    ->end()
                ->arrayNode('securityDefinitions')
                    ->info('Security scheme definitions that can be used
across the specification.')
                    ->ignoreExtraKeys(false)
                    ->children()
                    ->end()
                    ->end()
                ->arrayNode('security')
                    ->info('A declaration of which security schemes are
applied for the API as a whole.')
                    ->ignoreExtraKeys(false)
                    ->children()
                    ->end()
                    ->end();
    }
}

<?php

namespace Sushchyk\SymfonyRestDocs\Symfony\DependencyInjection;

use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\HttpKernel\DependencyInjection\Extension;

class RestDocsExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
        $configuration = $this->getConfiguration($configs, $container);
        $config = $this->processConfiguration($configuration, $configs);

        $container->setParameter('rest_docs.output', $config['output']);
        $container->setParameter('rest_docs.swagger', $config['swagger']);
    }
}

<?php
```

```
namespace Sushchyk\SymfonyRestDocs\Symfony;

use Symfony\Component\HttpKernel\Bundle\Bundle;

class RestDocsBundle extends Bundle
{

}


<?php

namespace Sushchyk\SymfonyRestDocs\Utils;

use Sushchyk\SymfonyRestDocs\Exception\DataConflictsWithSchemaException;
use Sushchyk\SymfonyRestDocs\OpenApi\Schema\Definitions;

class DataCompatibleWithSchemaValidator
{
    /**
     * @param array $schema
     *
     * @param $data
     * @param string $dataName
     *
     * @throws DataConflictsWithSchemaException
     */
    public function validate(array $schema, ?Definitions $definitions,
$data, $dataName = null)
    {
        if (isset($schema['$ref'])) {
            $schema = $definitions->getDefinition($schema['$ref']);
        }

        if ($schema['type'] === 'array') {
            if (!is_array($data)) {
                throw new DataConflictsWithSchemaException("Expecting %s to
be array", $dataName);
            }

            foreach ($data as $key => $item) {
                $this->validate($schema['items'], $definitions, $item,
"$dataName item #$key");
            }
        }

        if ($schema['type'] === 'object') {
            if (!is_array($data)) {
                throw new DataConflictsWithSchemaException("Expecting %s to
be object", $dataName);
            }

            foreach ($schema['required'] as $propertyName) {
                if (!isset($data[$propertyName])) {
                    throw new DataConflictsWithSchemaException("%s
`$propertyName` property is not defined", $dataName, false);
                }
                $this->validate($schema['properties'][$propertyName],
$definitions, $data[$propertyName], "$dataName property `$propertyName`");
            }
        }

        if ($schema['type'] === 'integer') {
```

```
            if (!is_integer($data)) {
                throw new DataConflictsWithSchemaException("Expecting %s to
be integer.", $dataName);
            }
        }

        if ($schema['type'] === 'string') {
            if (!is_string($data)) {
                throw new DataConflictsWithSchemaException("Expecting %s to
be string", $dataName);
            }
        }

        if ($schema['type'] === 'boolean') {
            if (!is_bool($data)) {
                throw new DataConflictsWithSchemaException("Expecting %s to
be boolean", $dataName);
            }
        }

    }
}
```

```
<?php

namespace Sushchyk\SymfonyRestDocs\Utils;

use Sushchyk\SymfonyRestDocs\OpenApi\Schema\Responses\Header;

class HeadersCreator
{
    public function createHeadersFromArray(array $headersArray)
    {
        $headers = [];

        foreach ($headersArray as $key => $header) {
            $headerDescription = $header['description'];
            unset($header['description']);

            $headerType = $header['type'];
            unset($header['type']);

            $headers[$key] = new Header($headerDescription, $headerType,
$header);
        }

        return $headers;
    }
}
```

```
<?php

namespace Sushchyk\SymfonyRestDocs\Utils;

use Sushchyk\SymfonyRestDocs\OpenApi\Schema\Paths\PathItem\Operation;

class OperationsMerger
{
    public function mergeOperations(Operation $firstOperation, Operation
$secondOperation): Operation
    {
        $resultOperation = new Operation($firstOperation->getUri(),
$firstOperation->getMethod());
```

```
        $headOperation = $firstOperation;
        if ($firstOperation->getMinResponseStatusCode() >=
$secondOperation->getMinResponseStatusCode()) {
            $headOperation = $secondOperation;
        }

        return $resultOperation;
    }
}

<?php

namespace Sushchyk\SymfonyRestDocs\Utils;

use Sushchyk\SymfonyRestDocs\OpenApi\Schema;
use Sushchyk\SymfonyRestDocs\OpenApi\Schema\Definitions;
use Sushchyk\SymfonyRestDocs\OpenApi\Schema\Info;
use Sushchyk\SymfonyRestDocs\OpenApi\Schema\Info\Contact;
use Sushchyk\SymfonyRestDocs\OpenApi\Schema\Info\License;
use Sushchyk\SymfonyRestDocs\OpenApi\Schema\ParametersDefinitions;
use
Sushchyk\SymfonyRestDocs\OpenApi\Schema\Paths\PathItem\Operation\Response;
use Sushchyk\SymfonyRestDocs\OpenApi\Schema\Paths\PathItem\Parameter;
use Sushchyk\SymfonyRestDocs\OpenApi\Schema\ResponsesDefinitions;
use Sushchyk\SymfonyRestDocs\OpenApi\Schema\Security\SecurityDefinition;
use Sushchyk\SymfonyRestDocs\OpenApi\Schema\SecurityDefinitions;

class SchemaCreatorFromArray
{
    /**
     * @var HeadersCreator
     */
    private $headersCreator;

    public function __construct()
    {
        $this->headersCreator = new HeadersCreator();
    }

    public function createSchemaFromArray(array $swaggerConfig): Schema
    {
        $swaggerSchema = new Schema();

        $swaggerSchema->setInfo($this->createInfo($swaggerConfig));

        if (isset($swaggerConfig['host'])) {
            $swaggerSchema->setHost($swaggerConfig['host']);
        }

        if (isset($swaggerConfig['schemes'])) {
            $swaggerSchema->setSchemes($swaggerConfig['schemes']);
        }

        if (isset($swaggerConfig['consumes'])) {
            $swaggerSchema->setConsumes($swaggerConfig['consumes']);
        }

        if (isset($swaggerConfig['produces'])) {
            $swaggerSchema->setProduces($swaggerConfig['produces']);
        }

        if (isset($swaggerConfig['definitions'])) {
            $swaggerSchema->setDefinitions(new
Definitions($swaggerConfig['definitions']));
```

```
        }

        if (isset($swaggerConfig['parameters'])) {
            $swaggerSchema->setParameters($this-
>createParametersDefinitions($swaggerConfig));
        }

        if (isset($swaggerConfig['responses'])) {;
            $swaggerSchema->setResponses($this-
>createResponsesDefinition($swaggerConfig));
        }

        if (isset($swaggerConfig['securityDefinitions'])) {
            $swaggerSchema->setSecurityDefinitions(
                $this-
>createSecurityDefinitions($swaggerConfig['securityDefinitions'])
            );
        }

        return $swaggerSchema;
    }

    private function createInfo(array $swaggerConfig): Info
    {
        $infoConfig = $swaggerConfig['info'];

        $contactConfig = $infoConfig['contact'] ?? null;
        $contact = $contactConfig
            ? new Contact($contactConfig['name'] ?? null,
$contactConfig['url'] ?? null, $contactConfig['email'] ?? null)
            : null;

        $licenseConfig = $infoConfig['license'] ?? null;
        $license = $licenseConfig
            ? new License($licenseConfig['name'] ?? null,
$licenseConfig['url'])
            : null;

        return new Info(
            $infoConfig['title'],
            $infoConfig['description'] ?? null,
            $infoConfig['termsOfService'] ?? null,
            $contact,
            $license,
            $infoConfig['version']
        );
    }

    private function createParametersDefinitions(array $swaggerConfig)
    {
        $parameters = [];
        foreach ($swaggerConfig['parameters'] as $key => $parameter) {
            $name = $parameter['name'];
            $in = $parameter['in'];
            unset($parameter['name']);
            unset($parameter['in']);
            $parameters[$key] = new Parameter($name, $in, $parameter,
null);
        }

        return new ParametersDefinitions($parameters);
    }

    private function createResponsesDefinition(array $swaggerConfig)
```

```
    {
        $responses = [];
        foreach ($swaggerConfig['responses'] as $response) {
            $headers = null;
            if (isset($response['headers'])) {
                $headers = $this->headersCreator-
>createHeadersFromArray($response['headers']);
            }

            $responses[] = new Response(
                1000,
                $response['description'],
                $response['schema'] ?? null,
                $headers
            );
        }

        return new ResponsesDefinitions($responses);
    }

    private function createSecurityDefinitions($securityDefinitions):
SecurityDefinitions
    {
        $definitions = [];

        foreach ($securityDefinitions as $key => $securityDefinitionArray)
{
            $type = $securityDefinitionArray['type'];
            unset($securityDefinitionArray['type']);

            $definitions[$key] = new SecurityDefinition($type,
$securityDefinitionArray);
        }

        return new SecurityDefinitions($definitions);
    }
}

<?php

namespace Sushchyk\SymfonyRestDocs\Utils;

use Sushchyk\SymfonyRestDocs\OpenApi\Schema;
use Sushchyk\SymfonyRestDocs\OpenApi\Schema\ExternalDocumentation;
use Sushchyk\SymfonyRestDocs\OpenApi\Schema\Paths;
use Sushchyk\SymfonyRestDocs\OpenApi\Schema\Paths\ParameterReference;
use Sushchyk\SymfonyRestDocs\OpenApi\Schema\Paths\PathItem\Operation;
use
Sushchyk\SymfonyRestDocs\OpenApi\Schema\Paths\PathItem\Operation\Response;
use Sushchyk\SymfonyRestDocs\OpenApi\Schema\Paths\PathItem\Parameter;
use Sushchyk\SymfonyRestDocs\OpenApi\Schema\SecurityRequirement;

class SchemaCreatorFromSwaggerFile
{
    /**
     * @var SchemaCreatorFromArray
     */
    private $schemaCreatorFromContainer;

    /**
     * @var HeadersCreator
     */
    private $headersCreator;
```

```
    public function __construct()
    {
        $this->schemaCreatorFromContainer = new SchemaCreatorFromArray();
        $this->headersCreator = new HeadersCreator();
    }

    public function createSchemaFromFile(string $pathToFile): Schema
    {
        $fileData = json_decode(file_get_contents($pathToFile), true);

        $schema = $this->schemaCreatorFromContainer-
>createSchemaFromArray($fileData);

        $paths = new Paths();
        foreach ($fileData['paths'] as $path => $pathOperations) {
            foreach ($pathOperations as $method => $operationArr) {
                $operation = new Operation($method, $path);

                if (isset($operationArr['tags'])) {
                    $operation->setTags($operationArr['tags']);
                }

                if (isset($operationArr['summary'])) {
                    $operation->setSummary($operationArr['summary']);
                }

                if (isset($operationArr['description'])) {
                    $operation-
>setDescription($operationArr['description']);
                }

                if (isset($operationArr['externalDocs'])) {
                    $operation->setExternalDocs(new ExternalDocumentation(
                        $operationArr['externalDocs']['url'] ?? null,
                        $operationArr['externalDocs']['description'] ??
null
                    ));
                }

                if (isset($operationArr['operationId'])) {
                    $operation-
>setOperationId($operationArr['operationId']);
                }

                if (isset($operationArr['produces'])) {
                    $operation->setProduces($operationArr['produces']);
                }

                if (isset($operationArr['consumes'])) {
                    $operation->setConsumes($operationArr['consumes']);
                }

                if (isset($operationArr['parameters'])) {
                    foreach ($operationArr['parameters'] as $parameter) {
                        if (isset($parameter['$ref'])) {
                            $operation->addParameter(new
ParameterReference($parameter['$ref'], null));
                        } else {
                            $name = $parameter['name'];
                            $in = $parameter['in'];
                            unset($parameter['name']);
                            unset($parameter['in']);
                            $operation->addParameter(new Parameter($name,
$in, $parameter, null));
```

```
                        }
                    }
                }

                if (isset($operationArr['security'])) {
                    $operation->setSecurity(new
SecurityRequirement($operationArr['security']));
                }

                if (isset($operationArr['deprecated'])) {
                    $operation->setDeprecated($operationArr['deprecated']);
                }

                foreach ($operationArr['responses'] as $code =>
$responseArr) {
                    $headers = null;
                    if (isset($responseArr['headers'])) {
                        $headers = $this->headersCreator-
>createHeadersFromArray($responseArr['headers']);
                    }

                    $response = new Response(
                        $code,
                        $responseArr['description'],
                        $responseArr['schema'] ?? null,
                        $headers
                    );

                    $operation->addResponse($code, $response);
                }

                $paths->addOperation($operation);
            }
        }

        $schema->setPaths($paths);

        return $schema;
    }
}

<?php

use PHPUnit\Framework\TestCase;
use Sushchyk\SymfonyRestDocs\Exception\DataConflictsWithSchemaException;
use Sushchyk\SymfonyRestDocs\Utils\DataCompatibleWithSchemaValidator;

class DataCompatibleWithSchemaValidatorTest extends TestCase
{
    private $dataBelongsToSchemaValidator;

    public function setUp(): void
    {
        $this->dataBelongsToSchemaValidator = new
DataCompatibleWithSchemaValidator();
    }

    /**
     * @dataProvider validationDataProvider
     *
     * @param $schema
     * @param $body
     * @param string|null $expectedError
     */
```

```
    public function testValidate($schema, $body, string $expectedError =
null)
    {
        $exception = null;
        try {
            $this->dataBelongsToSchemaValidator->validate($schema, null,
$body);
        }   catch (DataConflictsWithSchemaException $e) {
            $exception = $e;
        }

        if ($expectedError === null && $exception) {
            $this->assertTrue(false, 'Unexpected exception `' . $exception-
>getMessage() . '`` thrown');
        }

        if ($expectedError && !$exception) {
            $this->assertTrue('Failed assert that exception was thrown');
        }

        if ($expectedError && $exception) {
            $this->assertEquals($expectedError, $exception->getMessage());
        }

        $this->assertEquals(true, true);
    }

    public function validationDataProvider()
    {
        $arrayOfTasksSchema = [
            'type'  => 'array',
            'items' => [
                'type'       => 'object',
                'properties' => [
                    'id'          => [
                        'type' => 'integer'
                    ],
                    'title'       => [
                        'type' => 'string'
                    ],
                    'description' => [
                        'type' => 'string'
                    ]
                ],
                'required'   => [
                    'id',
                    'title',
                    'description'
                ]
            ]
        ];


        return [
            [$arrayOfTasksSchema, [['id' => 1, 'title' => 'Task 1 Title',
'description' => 'Task 3']]],
            [$arrayOfTasksSchema, []],
            [
                $arrayOfTasksSchema,
                [['id' => 1, 'title' => 'Task 2 Title']],
                ' item #0 `description` property is not defined'
            ],
            [
                $arrayOfTasksSchema,
```

```
                [['id' => "1", 'title' => 'Task 1 Title', 'description' =>
'Task 3']],
                'Expecting  item #0 property `id` to be integer.'
            ],
            [
                $arrayOfTasksSchema,
                [['id' => 1, 'title' => 'Task 2 Title', 'description' =>
'Task 3'], []],
                ' item #1 `id` property is not defined'
            ],
            [$arrayOfTasksSchema, null, 'Expecting data to be array'],
            [$arrayOfTasksSchema, '', 'Expecting data to be array'],
            [$arrayOfTasksSchema, false, 'Expecting data to be array'],
            [$arrayOfTasksSchema, 2444, 'Expecting data to be array'],
            [['type' => 'boolean'], 24, 'Expecting data to be boolean'],
            [['type' => 'boolean'], false],
        ];
    }
```

Додаток 2

Копія презентації

МЕТОД СТВОРЕННЯ ДОКУМЕНТАЦІЇ ДЛЯ REST API НА ОСНОВІ ТЕСТІВ

Магістрант: Сущик Андрій Миколайович
Науковий керівник: Олещенко Любов Михайлівна

Актуальність дослідження

Програмісти витрачають відносно небагато часу на написання коду. Більшість часу йде на аналіз існуючого коду, аналіз доменної логіки, а також написання тестів та документації.

Предмет дослідження

Методи створення документації для REST API.

Об’єкт дослідження

Процес створення документації для REST API.

Мета дослідження

оптимізація та автоматизація процесу створення документації для REST API.

Стандарти для API

GraphQL

JSON RPC

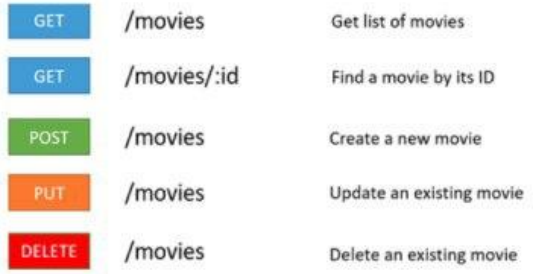
SOAP

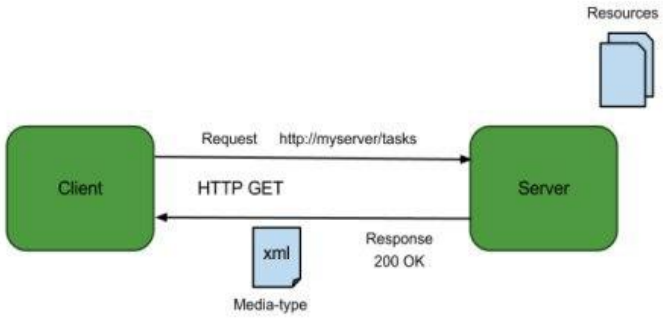
REST

{ REST }

REST API

- Ключове поняття в REST - це ресурс. Ресурс має стан, і ми можемо його отримувати або змінювати.
- Найбільш широко поширений стандарт.
- Клієнт-серверна архітектура, відсутність стану, одноматність інтерфейсу.





Вимоги до документації для REST API

Актуальність	Інтерактивність	Швидкість
Документація завжди повинна бути в актуальному стані.	Документація повинна надавати “пісочницю”, де можна робити запити до серверної частини.	Мінімальна трата часу на створення та підтримку документації.

Методи створення документації для REST API

- за допомогою бібліотек/інструментів для конкретної мови програмування/фреймворку;
- написання вручну;
- за допомогою сторонніх утиліт (Postman);
- на основі тестів.

Створення документації за допомогою бібліотек/інструментів для конкретної мови програмування

coreapi для Django;
spring-rest-docs для Spring;
laravel-docs для Laravel.

ПЕРЕВАГИ

- дозволяють суттєво зекономити час;
- дозволяють не винаходити “велосипед”;

НЕДОЛІКИ

- “засмічують” код анотаціями та файлами;
- необхідність підтримки документації у актуальному стані;

Створення документації вручну

ПЕРЕВАГИ

- документація буде більш зрозумілою для користувачів API;
- немає прив’язки до платформи/мови програмування;

НЕДОЛІКИ

- потребує багато часу;
- необхідність підтримки документації у актуальному стані;

Створення документації за допомогою сторонніх утиліт

Postman;
RESTful Menu;
Paw 3.

ПЕРЕВАГИ

- немає прив’язки до платформи/мови програмування;
- зазвичай, такі утиліти надають графічний інтерфейс;

НЕДОЛІКИ

- “засмічують” код анотаціями та файлами;
- необхідність підтримки документації у актуальному стані;

Створення документації на основі тестів

spring-rest-docs

ПЕРЕВАГИ

- документація завжди в актуальному стані;
- змушує розробників писати функціональні тести для свого коду;

НЕДОЛІКИ

- зв’язаність тестів та документації;

Результати аналізу методів

Критерій / метод	створення документації за допомогою інструментів для конкретної мови програмування/ фреймворку	створення документації вручну	створення документації за допомогою сторонніх утиліт	створення документації на основі тестів
Забезпечує актуальність документації	-/+	-	-	+
Надає графічний інтерфейс	+/-	-	+	+
Не потребує суттєвих часових затрат	+	-	+/-	-/+

Spring Rest Docs. Принцип роботи

Spring Rest Docs. Недоліки

- необхідність вносити зміни в шаблони;
- документація не є інтерактивною;
- функціональні обмеження.

Запропонований метод

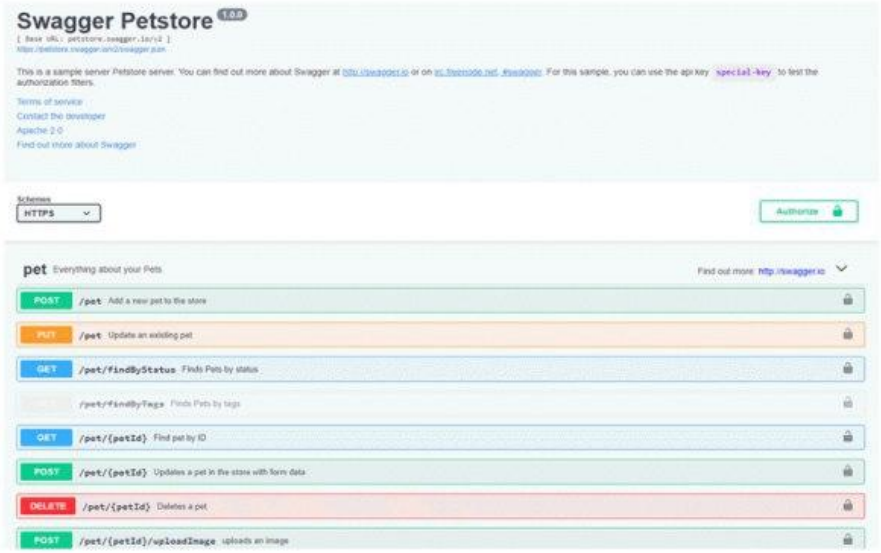
Було запропоновано новий метод на основі тестів, якому не властиві недоліки Spring Rest Docs.

Ключовою особливістю даного методу є використання специфікації Open API, відомої як Swagger.

Запропонований метод. Принцип роботи

Результатом запуску тестів є Swagger-файл.

Для візуалізації потрібно використовувати бібліотеку “swagger-ui“.



Використані технології

Метод реалізований у вигляді бібліотеки для пакетного менеджера Composer.

Бібліотека сумісна з версіями PHP >= 7.1 та Symfony >= 4.0

Інтерфейс методу

Метод надає можливість задати параметри вихідної документації шляхом використання конфігураційного файлу в yaml-форматі, а безпосередньо в тестах необхідно використовувати клас Request.

```
(new Request('GET', '/api/tasks'))
      ->withSummary('Get tasks list')
      ->expectsResponse(200, [
            'description' => 'Returns list of tasks',
            'schema' => [
                  'type' => 'object',
                  'required' => ['tasks'],
                  'properties' => [
                        'tasks' => [
                        'type' => 'array',
                        'items' => [
                              'type' => 'object',
                              'required' => [
                                    'Id', 'title', 'description'
                                    ...
            ));
```

```
output: 'public/assets/swagger.json'
swagger:
      info: `App Documentation
      version: '2.0'
```

Тестування

Для тестування використовувалась бібліотека PHPUnit.

65% коду покриті автоматичними інтеграційними та модульними тестами.

Оцінка ефективності

Для оцінки ефективності був написаний тестовий проект.

Даний метод потребує на 25% менше рядків коду ніж при використанні NelmioDocApiBundle.

Даний метод забезпечує інтерактивність документації.

Висновки

В ході роботи над магістерською дисертацію проаналізовані стандарти для API, методи для створення документації для REST API, сформовані вимоги до документації.

Проаналізовано наявну реалізацію методу на основі тестів Spring Rest Docs та запропоновано власний метод.

Проведено тестування та оцінку ефективності даного методу.

ДЯКУЮ ЗА УВАГУ!